Задача 1. Множество целых чисел

Источник:	базовая I
Имя входного файла:	`---`
Имя выходного файла:	`---`
Ограничение по времени:	3 секунды*
Ограничение по памяти:	разумное

Код задачи: `4/1_integerset`

Требуется написать структуру данных для хранения множества целых чисел, которая позволяет быстро узнать, лежит ли заданное число в множестве. Структуру данных надо оформить в виде динамической библиотеки.

В систему проверки требуется отправить:
- Скрипт сборки `c.bat.txt` для Windows.
- Скрипт сборки `c.sh.txt` для Linux.
- Хедер `integerset.h`.
- Любые дополнительные файлы хедеров и исходного кода.

Проверка в этой задаче проводится и под Windows, и под Linux. Скрипт для Windows должен содержать последовательность команд для сборки динамической библиотеки `integerset.dll` вместе с библиотекой импорта `integerset.lib`. Тестирующий код жюри будет собираться с созданной библиотекой импорта. Скрипт для Linux должен собирать динамическую библиотеку `integerset.so`, с которой будет собираться тестирующий код жюри.

В хедере `integerset.h` должен быть объявлен тип `IntegerSet`, а также две функции `CreateSet` и `IsInSet`. Сигнатуру этих функций и принципы работы следует определить самостоятельно, исходя из примера использования, приведённого на следующей странице.

Гарантируется, что размер создаваемого множества всегда будет неотрицательным. Также гарантируется, что в `IsInSet` будет передаваться либо указатель, который ранее вернула функция `CreateSet`, либо ноль. Количество вызовов каждой функции не будет превышать $3 \cdot 10^5$. Размер каждого множества не будет превышать 10^5. Сумма размеров всех созданных множеств не будет превышать $3 \cdot 10^5$. Освобождать память из под множеств в этой задаче **не** нужно.

Пример кода, который должен собираться вместе с библиотекой импорта и работать:

```
#include "integerset.h"
#include <assert.h>
#include <limits.h>

int main() {
    int myarr[] = {1, 2, 3};
    //create set with numbers 1, 2, and 3
    IntegerSet *s123 = CreateSet(myarr, 3);
    myarr[1] = -5;
    assert(IsInSet(s123, 3) == 1 && IsInSet(s123, 2) == 1);
    assert(IsInSet(s123, 0) == 0 && IsInSet(s123, 4) == 0);
    //create set with numbers -5 and 3
    IntegerSet *s15 = CreateSet(myarr + 1, 2);
    assert(IsInSet(s15, 3) == 1 && IsInSet(s15, -5) == 1);
    assert(IsInSet(s15, 1) == 0 && IsInSet(s15, 2) == 0);
    //create empty set (note: null pointer is allowed only when size is 0)
    IntegerSet *sEmpty = CreateSet(0, 0);
    assert(sEmpty != 0);
    assert(IsInSet(sEmpty, 0) == 0 && IsInSet(sEmpty, -123456789) == 0);
    IntegerSet *sNull = 0;
    //null pointer must be treated by IsInSet as empty set
    assert(IsInSet(sNull, 0) == 0 && IsInSet(sNull, -123456789) == 0);
    myarr[0] = -5;
    //if array has equal elements, then CreateSet must return 0
    IntegerSet *sBad = CreateSet(myarr, 3);
    assert(sBad == 0);
    int largearr[] = {1, 5, INT_MAX, INT_MIN, 1000000000, -123, 555, 0};
    //create set with 8 numbers from largearr
    IntegerSet *sLarge = CreateSet(largearr, 8);
    assert(IsInSet(sLarge, INT_MAX) == 1 && IsInSet(sLarge, INT_MIN) == 1);
    assert(IsInSet(sLarge, 1000000000) == 1 && IsInSet(sLarge, -123) == 1);
    assert(IsInSet(sLarge, 123) == 0 && IsInSet(sLarge, -5) == 0);
    largearr[1] = 5;
    sLarge = CreateSet(largearr, 8); //same set as previously
    assert(IsInSet(sLarge, 5) == 1);
    largearr[7] = 5;
    sLarge = CreateSet(largearr, 8); //now it has equal elements
    assert(sLarge == 0);
    return 0;
}
```

Задача 2. BLAS в квадрате

Источник:	базовая I
Имя входного файла:	--
Имя выходного файла:	--
Ограничение по времени:	10 секунд*
Ограничение по памяти:	разумное

Код задачи: `4/2_blassqr`

В этой задаче требуется вычислить матрицу в квадрате, используя функцию `dgemm` из настоящей библиотеки OpenBLAS. Полный набор файлов этой библиотеки, которые будут доступны при проверке, можно скачать здесь. Для использования следует подключить хедер `cblas.h` с интерфейсом, предназначенным для языка C. Подробную документацию функции `cblas_dgemm` можно посмотреть на сайте Intel здесь. Учтите, что тип `MKL_INT` — это на самом деле просто тип `int`.

Кроме того, необходимо написать собственную реализацию функции `cblas_dgemm` и собрать из неё динамическую библиотеку, которая будет бинарно совместимой с реализацией от OpenBLAS.

Часть 1. Используя библиотеку OpenBLAS, реализуйте в файле `matrixsqr.c` функцию:

```
// Computes R = A * A;
// Here A and R are square matrices N x N.
// Every matrix is stored in row-major layout,
// i.e. A[i*n+j] is the element in i-th row and j-th column.
void MatrixSqr(int n, const double *A, double *R);
```

Пример использования этой функции (содержимое файла `main.c`):

```
#include <assert.h>
void MatrixSqr(int n, const double *A, double *R);
int main() {
    double A[3][3] = {{0,1,2},{2,0,0},{3,0,1}};
    double R[3][3];
    MatrixSqr(3, &A[0][0], &R[0][0]);
    assert(R[0][0] == 8 && R[0][1] == 0 && R[0][2] == 2);
    assert(R[1][0] == 0 && R[1][1] == 2 && R[1][2] == 4);
    assert(R[2][0] == 3 && R[2][1] == 3 && R[2][2] == 7);
    return 0;
}
```

В ходе проверки файл `main.c` будет подменяться на тестовый код жюри, чтобы проверить правильность работы вашей `MatrixSqr`. От вас требуется собрать из `main.c` и `matrixsqr.c` исполняемый файл `checked_solution.exe` (если под Linux, то без расширения).

Часть 2. Напишите собственную простую реализацию функции `cblas_dgemm` с такой же сигнатурой, как у функции из OpenBLAS. Можно заменить enum-типы `CBLAS_LAYOUT` и `CBLAS_TRANSPOSE`, а также тип `MKL_INT`, на тип `int`. Учтите, что от вашей реализации **не** требуется поддерживать все многочисленные настройки, которые есть в настоящем BLAS. Достаточно, чтобы написанный вами код `MatrixSqr` работал с использованием вашей реализации `cblas_dgemm` так же правильно, как с использованием реализации из OpenBLAS.

Функцию следует реализовать в файле `myblas.c`.

При проверке на Windows из этого файла будет собираться DLL с помощью командной строки:

```
cl myblas.c /LD /O2 /link/EXPORT:cblas_dgemm,@6491
```

К сожалению, библиотека OpenBLAS использует сомнительный способ экспортирования "по ординалам". Последний аргумент командной строки указывает, что нужно экспортировать функцию `cblas_dgemm` с заданным ординалом 6491. Это необходимо для достижения бинарной совместимости с OpenBLAS — то есть чтобы можно было подменить одну DLL другой.

При проверке на Linux SO будет собираться обычным способом:

```
gcc myblas.c -shared -fPIC -O2 -o myblas.so
```

Учтите, что эта библиотека будет лежать в рабочей директории, так что при сборке исполняемого файла нужно прописать текущую директорию в пути загрузчика.

В данной задаче вы можете **выбрать** между проверкой на Windows и на Linux. На проверку требуется отправить:

- `matrixsqr.c` — содержит реализацию функции `MatrixSqr`.
- `myblas.c` — содержит собственную реализацию функции `cblas_dgemm`.
- `c.bat.txt` — скрипт сборки, если хочется проверку на Windows.
- `c.sh.txt` — скрипт сборки, если хочется проверку на Linux.

Во время сборки и запуска в текущей директории будут доступны:

- `main.c` — файл с тестовым кодом жюри, в котором есть функция `main`.
- Набор файлов библиотеки OpenBLAS (см. выше).

Проверка будет выполняться следующим образом:

1. Будет запущен ваш скрипт сборки (`c.bat.txt` под Windows), который должен собрать исполняемый файл `checked_solution.exe`.
2. Будет запущена программа `checked_solution.exe` на наборе тестов ($n \leqslant 1\,500$).
3. Будет собрана DL из `myblas.c`, которая будут переименована в файл `libopenblas.dll` (или `.so`) с заменой последнего.
4. Будет снова запущена программа `checked_solution.exe` на наборе тестов ($n \leqslant 700$).

Заметьте, что перед шагом 3 исполняемый файл **не** пересобирается: собранная программа должна отработать так же хорошо с подменённой DL. Правда к вашему решению гораздо слабее требования по скорости: всё-таки, реализация от OpenBLAS работает намного быстрее.

Задача 3. Списки инцидентности

Источник:	базовая II
Имя входного файла:	--
Имя выходного файла:	--
Ограничение по времени:	3 секунды*
Ограничение по памяти:	128 мегабайт

Код задачи: `4/3_edgelists`

Дан неориентированный граф из N вершин и M рёбер. Рёбра заданы одним списком, каждое ребро задаётся парой концевых вершин и весом. Возможны кратные рёбра и петли. Требуется за один проход по списку рёбер составить списки инцидентности для всех вершин.

От вас требуется отправить один файл `sol.c`. В нём следует подключить хедер `sol.h` следующего содержания:

```
#ifndef EDGELISTS_SOL_579846984654
#define EDGELISTS_SOL_579846984654

typedef struct {
    int from, to;
    int weight;
} Edge;

//================================================
//call these functions to get the graph data:

//returns N -- the number of vertices in the graph
int getVerticesCount();
//reads the next edge from the edge list
//if there is next edge, returns 1 and stores that edge to *oEdge
//if there is no next edge, returns 0 without touching pointer
int readEdge(Edge *oEdge);

//================================================
//implement these functions in your solution:

//this function is called first to initialize graph
//you are expected to read graph here and fill internal data structures
void init();

//returns number of edges indicent to vertex iVertex
int getEdgesCount(int iVertex);
//returns iIndex-th edge incident to the vertex iVertex
//it must have .from == Vertex and .to denoting the other end
Edge getIncidentEdge(int iVertex, int iIndex);

#endif
```

В `sol.c` нужно реализовать функции `init`, `getEdgesCount` и `getIncidentEdge`. Функции `getVerticesCount`, `readEdge` будут реализованы в другой единице трансляции и слинкованы вместе с вашим кодом. Для тестирования “у себя” вам желательно тоже их где-то реализовать, но отправлять в систему их **не** нужно.

Ограничения на размер графа: $N \leqslant 3 \cdot 10^5$, $M \leqslant 3 \cdot 10^5$. В данной задаче всё нумеруется начиная с нуля.

Гарантируется, что при тестировании функция `init` будет вызвана один раз до всех остальных вызовов. Остальные две функции могут вызываться в произвольном порядке и объёме. Гарантируется, что все вызовы корректны: `iVertex` лежит в пределах от 0 до $(N-1)$, а `iIndex` в пределах от 0 до того, что вернула ваша функция `getEdgesCount`, минус один. Всего количество вызовов `getEdgesCount` и `getIncidentEdge` не превышает 10^6.

Обратите внимание, что каждое ребро должно входить в список инцидентности обеих своих концевых вершин. Петля должна входить дважды в список рёбер своей вершины. Порядок рёбер в каждом списке инцидентности значения не имеет.

Пример

Ниже схематично приведён порядок вызовов функций на первом тесте. Для кратности `getIncidentEdge` обозначается как `getIE`.

--	--
5 = getVerticesCount() 1 = readEdge() : [0, 2, 178] 1 = readEdge() : [3, 4, 207] 1 = readEdge() : [1, 1, 356] 1 = readEdge() : [2, 0, 101] 1 = readEdge() : [4, 1, 286] 1 = readEdge() : [4, 1, 213] 0 = readEdge() 0 = readEdge()	init() 2 = getEdgesCount(0) [0, 2, 178] = getIE(0, 0) [0, 2, 101] = getIE(0, 1) 4 = getEdgesCount(1) [1, 1, 356] = getIE(1, 0) [1, 4, 286] = getIE(1, 3) [1, 1, 356] = getIE(1, 2) [1, 4, 213] = getIE(1, 1) 2 = getEdgesCount(2) [2, 0, 178] = getIE(2, 0) [2, 0, 101] = getIE(2, 1) 1 = getEdgesCount(3) [3, 4, 207] = getIE(3, 0) 3 = getEdgesCount(4) [4, 3, 207] = getIE(4, 2) [4, 1, 286] = getIE(4, 0) [4, 1, 213] = getIE(4, 1)

Задача 4. core DL

Источник:	основная I
Имя входного файла:	`---`
Имя выходного файла:	`stdout`
Ограничение по времени:	разумное
Ограничение по памяти:	разумное

Код задачи: `4/4_core`

Пусть `State` — это массив из 256 строковых “регистров”:

```
typedef struct State {
    char *regs[256];
} State;
```

Каждый регистр либо нулевой (т.е. указатель равен 0), либо указывает на C-шную строку, расположенную на куче.

Требуется реализовать следующие функции:

```
//prints 'ECHO: ' and all passed strings separated by '|'
void echo_0(State *state);
void echo_1(State *state, char *arg0);
void echo_2(State *state, char *arg0, char *arg1);
void echo_3(State *state, char *arg0, char *arg1, char *arg2);
//prints contents of I-th register (it must not be NULL)
//[idx] contains decimal representation of I
void print_1(State *state, char *idx);
//prints all non-NULL registers with their values (sorted by register number)
void printregs_0(State *state);
//saves a copy of string [what] into I-th register
//[idx] contains decimal representation of I
void store_2(State *state, char *idx, char *what);
//copies contents of S-th register into D-th register (S-th register is not NULL)
//[dst] and [src] contain decimal representations of D and S respectively
//BEWARE: [dst] and [src] are allowed to be equal indices
void copy_2(State *state, char *dst, char *src);
//assigns NULL to I-th register
//[idx] contains decimal representation of I
void clear_1(State *state, char *idx);
```

Замечания:

1. Требуемый формат вывода у функций `echo_*`, `print_1` и `printregs_0` можно посмотреть в примере ниже.
2. Функции `echo_*` хоть и принимают параметр `state`, но его **не** используют.
3. Учтите, что каждый ненулевой регистр должен указывать на собственный буфер памяти, ни с кем не разделённый. Если функция записывает что-то в регистр, то память от старого значения нужно удалить, а память для нового значения нужно выделить.

Если вы хотите, чтобы проверка выполнялась на Windows, отправьте в систему тестирования исходный код и скрипт сборки `c.bat.txt`. Скрипт должен содержать последовательность команд для сборки динамической библиотеки `core.dll` вместе с библиотекой импорта. Для проверки на Linux отправьте исходный код и скрипт сборки `c.sh.txt`. Скрипт должен собрать динамическую библиотеку `core.so`.

Код примера:

```
#include <assert.h>
typedef struct State {
    char *regs[256];
} State;
#include "decls.h"  //contains function declarations (chunk of code above)
State state;
int main() {
    echo_2  (&state, "hello", "world");
    echo_0  (&state);
    echo_1  (&state, "the_only_argument");
    echo_3  (&state, "a", "b", "c");
    store_2 (&state, "13", "thirteen");
    store_2 (&state, "10", "ten");
    store_2 (&state, "15", "fifteen");
    store_2 (&state, "20", "twelve");
    echo_1  (&state, "==state==");
    printregs_0(&state);
    echo_1  (&state, "==copying==");
    print_1 (&state, "13");
    print_1 (&state, "15");
    copy_2  (&state, "13", "15");
    print_1 (&state, "13");
    print_1 (&state, "15");
    echo_1  (&state, "==clear==");
    clear_1 (&state, "10");
    clear_1 (&state, "15");
    store_2 (&state, "13", "thirteen_V2");
    printregs_0(&state);
}
```

В результате запуска в стандартный поток вывода должно быть записано:

```
ECHO: hello|world
ECHO:
ECHO: the_only_argument
ECHO: a|b|c
ECHO: ==state==
10 = ten
13 = thirteen
15 = fifteen
20 = twelve
ECHO: ==copying==
thirteen
fifteen
fifteen
fifteen
ECHO: ==clear==
13 = thirteen_V2
20 = twelve
```

Задача 5. Плагины

Источник:	основная I
Имя входного файла:	`stdin`
Имя выходного файла:	`stdout`
Ограничение по времени:	разумное
Ограничение по памяти:	разумное

Код задачи: `4/5_plugins`

В продолжение предыдущей задачи, предлагается реализовать минимальный строковый "ассемблер", команды которого загружаются из плагинов — динамических библиотек.

У процессора есть состояние `State` — это массив из 256 строковых "регистров".

```
typedef struct State {
    char *regs[256];
} State;
```

Каждый регистр либо нулевой (т.е. указатель равен 0), либо указывает на C-шную строку, расположенную на куче.

На стандартный поток ввода поступают команды, их все нужно выполнить в порядке их поступления. Каждая строка ввода — это одна команда, полная длина команды не превышает 1000 символов. Команда состоит из слов, разделённых проблеми (внутри слов пробелов нет). Первое слово в команде — это имя команды, а остальные слова (от нуля до трёх штук включительно) — это аргументы.

Код реализации команд разбросан по динамическим библиотекам, "плагинам". Имя команды записывается в формате: сначала имя плагина, потом двоеточие, потом имя функции. Как имя плагина, так и имя функции — это строки длиной не более 30 символов, состоящие только из латинских букв и цифр. Имя плагина может быть опущено: в таком случае имя команды состоит лишь из имени функции, а имя плагина считается автоматически равным "`core`".

Чтобы выполнить команду, ваша программа должна загрузить динамическую библиотеку из файла, имя которого совпадает с именем плагина (расширение `.dll` или `.so`). После этого нужно найти в этой библиотеке функцию, имя которой составляется как: имя функции, подчерк, количество аргументов. Далее нужно запустить эту функцию, передав ей сначала указатель на `State` (глобальное состояние програмы), а затем указатели на аргументы команды.

Рассмотрим примеры:

- `core:echo hello world`

  Здесь имя команды `core:echo`, имя плагина `core`, имя функции `echo`, два аргумента `hello` и `world`. Нужно выполнить функцию с именем `echo_2` из динамической библиотеки `core.dll`:

  ```
  //arg0 = 'hello', arg1 = 'world'
  echo_2(&state, arg0, arg1);
  ```

- `echo a b c`

  Здесь имя команды `echo`, значит имя плагина по умолчанию `core`. Нужно вызвать из `core.dll` функцию `echo_3`, передав все три строки как аргументы:

```
//arg0 = 'a', arg1 = 'b', arg2 = 'c'
echo_3(&state, arg0, arg1, arg2);
```

- `string:tokenize mama_mila__ramu_`
  `printregs`

  Здесь две команды. В первой нужно выполнить функцию `tokenize_1` из `string.dll`, передав ей указанную строку как аргумент. Во второй нужно выполнить `printregs_0` из `core.dll`.

Наверное вы уже заметили, что библиотека `core.dll` из первой задачи отлично работает в качестве плагина к этой задаче.

Кроме того, в этой задаче требуется реализовать второй плагин `string.dll` со следующими функциями:

```
//loads string A from I-th register ([idx0] contains its index)
//loads string B from J-th register ([idx1] contains its index)
//then stores concatenation of A and B into I-th register
//BEWARE: [idx0] and [idx1] are allowed to be equal indices
void concat_2(State *state, char *idx0, char *idx1);
//extracts sequence of tokens/words separated by underscore character from string [arg]
//puts K -- number of tokens into 0-th register
//puts the tokens into 1-th, 2-th, ..., K-th registers (in correct order)
void tokenize_1(State *state, char *arg);
```

Кроме того, от вас требуется собственно реализовать строковый ассемблер — исполняемый файл, работающий по описанным правилам. Он должен обрабатывать ошибку при загрузке функции:

1. Если для выполняемой команды нет файла динамической библиотеки с заданным именем, надо выдать сообщение "`Missing plugin имяплагина`". Здесь имя плагина берётся без расширения `.dll` (или `.so`).
2. Если файл плагина есть, но в нём нет нужной функции, нужно выдать сообщение "`Missing function имяфункции in plugin имяплагина`". Здесь имя функции должно включать подчёркивание и количество аргументов.

Других проблем при тестировании **не** будет.

При тестировании будут использоваться в том числе плагины жюри. Все имеющиеся в наличии плагины будут находиться в рабочей директории, там же будет запускаться ваша программа.

В ходе запуска одного теста будет использовано не более 35 различных плагинов. Количество выполненных команд в каждом тесте не превышает 20 000.

Требуется отправить в систему тестирования набор исходных файлов, а также два скрипта сборки. Скрипт `string.bat.txt` должен собирать динамическую библиотеку `string.dll` с указанными выше двумя функциями. Скрипт `myasm.bat.txt` должен собирать исполняемый файл `myasm.exe` строкового ассемблера.

Для проверки на Linux скрипты должны иметь расширение `.sh.txt`, библиотеки — `.so`, а исполняемый файл должен быть без расширения.

Пример

В примере полагается, что есть только два плагина: `core.dll` из задачи 1 и `string.dll`, описанный выше.

stdin

`string:tokenize mama_mila__ramu_`
`printregs`
`concat 1 2`
`string:concat 1 2`
`clear 3`
`printregs`
`string:concat 2 1`
`print 2`
`omg:sos 666`

stdout

`0 = 3`
`1 = mama`
`2 = mila`
`3 = ramu`
`Missing function concat_2 in plugin core`
`0 = 3`
`1 = mamamila`
`2 = mila`
`milamamamila`
`Missing plugin omg`

Комментарий

Обратите внимание, что поведение программы при запуске зависит от многих факторов. Например, собран Debug, Release или ручная сборка из командной строки, запущена программа с отладкой Visual Studio или напрямую, как подаются данные на вход и вытягиваются на выходе.

Если вы получаете Wrong Answer на примере, но локально у вас пример работает, рекомендуется сделать так:

1. Соберите программу и библиотеку, запустив свои .bat-файлы.
2. Создайте файл `input.txt` и скопируйте в него входные данные из примера.
3. Запустите в командной строке: `myasm.exe <input.txt >output.txt`
4. Проверьте полученный вывод в файле.

Знак “меньше” в консоли перенаправляет содержимое заданного файла в stdin. Аналогично работает знак “больше” с stdout.

Задача 6. Хеш-таблица

Источник:	основная I
Имя входного файла:	---
Имя выходного файла:	---
Ограничение по времени:	разумное
Ограничение по памяти:	разумное

Код задачи: `4/6_hashmap`

В этой задаче нужно реализовать "map" (отображение) с помощью хеш-таблицы.

По сути, отображение — это функция, отображающая «ключи» в «значения», и которую можно изменять. В map хранится набор пар «ключ-значение», у всех пар всегда разные ключи. Можно выполнять операции: установить заданное значение для заданного ключа, узнать значение для заданного ключа.

Например, рассмотрим отображение целочисленных ключей в строковые значения:

$$5 \to five \quad 3 \to three \quad 13 \to bignumber \quad 17 \to bignumber \quad 7 \to seven$$

Если установить для ключа 6 значение six, а для ключа 3 значение $triple$, то отображение примет вид:

$$5 \to five \quad 3 \to triple \quad 13 \to bignumber \quad 17 \to bignumber \quad 7 \to seven \quad 6 \to six$$

Чтобы быстро находить элемент в отображении по заданному ключу, вам нужно реализовать хеш-таблицу. В хеш-таблице хранятся лишь указатели на ключ и на значение, которые задаёт пользователь. Чтобы хеш-таблицу можно было применять для любых типов ключей и значений, эти указатели нетипизированные (`void*`, точнее `const void*`). Сами ключи и значения хранит пользователь: он обеспечивает их хранение. Гарантируется, что каждые занесённые в хеш-таблицу ключ и значение остаются валидными как минимум пока их не удалят из хеш-таблицы.

Очевидно, ключи разного типа сравниваются между собой по-разному. Поэтому пользователь хеш-таблицы задаёт указатель на функцию `EqualFunc`, с помощью которых хеш-таблица может проверить для любых двух ключей, совпадают ли они. Кроме того, для хеш-таблицы необходима хеш-функция, которая также задаётся пользователем (`HashFunc`). Гарантируется, что эти функции корректны: `EqualFunc` является эквивалентностью (есть транзитивность, симметричность и рефлексивность), `HashFunc` работает детерминированно и всегда выдаёт одинаковый хеш для равных ключей. Функции `EqualFunc` и `HashFunc` можно вызывать для всех ключей, которые сейчас лежат в таблице, а также для ключа, который передан явно в выполняемую функцию. Естественно, вызывать их для нулевого указателя или для указателя на уже удалённый из таблицы ключ нельзя.

При создании хеш-таблицы пользователь задаёт количество ячеек в ней. Пользователь гарантирует, что в хеш-таблице всегда будет хотя бы одна свободная ячейка. Более того, если вы используйте открытую адресацию (линейное пробирование и т.п.), то степень заполненности таблицы будет ограничена разумной константой. Хеш-функция возвращает 32-битное число, которое достаточно хорошо распределено на всему диапазону от 0 до $2^{32} - 1$. Благодаря этим условиям ваша хеш-таблица должна работать достаточно быстро. Естественно, вы можете реализовать разрешение коллизий через цепочки: даже в этом случае рекомендуется заводить предписанное количество ячеек в таблице.

Вы должны оформить вашу хеш-таблицу как динамическую библиотеку. В хедере `hashmap.h` должна быть объявлена структура `HashMap`, а также следующие типы и функции:

```
//pointer to key or value (untyped)
typedef const void *cpvoid;
//returns 1 if and only if two keys pointed by [a] and [b] are equal
//returns 0 otherwise
typedef int (*EqualFunc)(cpvoid a, cpvoid b);
//returns 32-bit hash of a key pointed by [key]
typedef uint32_t (*HashFunc)(cpvoid key);

//creates and returns new hash table with:
// [ef] -- function which compares keys for equality
// [hf] -- function which produces a hash for a key
// [size] -- prescribed size/capacity of the hash table (number of cells)
HashMap HM_Init(EqualFunc ef, HashFunc hf, int size);
//frees memory of hash map [self]
//note: called exactly once for every hash map created by HM_Init
void HM_Destroy(HashMap *self);

//returns value corresponding to the specified key [key] in hash map [self]
//if [key] is not present in the map, then returns NULL
cpvoid HM_Get(const HashMap *self, cpvoid key);
//sets value [value] for the key [key] in hash map [self]
//if [self] already has some value for [key], it is overwritten
void HM_Set(HashMap *self, cpvoid key, cpvoid value);
```

Кроме хедера `hashmap.h` вы можете отправить любое количество файлов исходного кода или хедеров.

Проверка будет выполняться и под Windows, и под Linux. Все файлы исходного кода будут собираться в `hashmap.dll` или `hashmap.so` с указанием макроса `HASHMAP_EXPORTS`. Этот макрос можно использовать для экспортирования функций в общепринятом порядке. Далее тестирующий код жюри будет собираться с созданной библиотекой импорта `hashmap.lib` (или с `hashmap.so` в случае Linux).

Пример кода, который должен собираться вместе с библиотекой и работать:

```
#include "hashmap.h"
#include <assert.h>
#include <string.h>

#define INTOF(ptr) (*(int*)(ptr))
int IntEqualFunc(cpvoid a, cpvoid b) {
    return (INTOF(a) - INTOF(b)) % 1000 == 0;
}
uint32_t IntHashFunc(cpvoid key) {
    int t = INTOF(key) % 1000;
    if (t < 0)
        t += 1000;
    return t * 0xDEADBEEF;
}

int main() {
    int data[] = {13, 174, 1013, -987, 0, 1};

    HashMap h = HM_Init(IntEqualFunc, IntHashFunc, 5);
    HM_Set(&h, &data[0], "hello");
    HM_Set(&h, &data[1], "world");
    assert(strcmp(HM_Get(&h, &data[2]), "hello") == 0);
    assert(strcmp(HM_Get(&h, &data[3]), "hello") == 0);
    HM_Set(&h, &data[4], "zero");
    assert(strcmp(HM_Get(&h, &data[4]), "zero") == 0);
    assert(strcmp(HM_Get(&h, &data[1]), "world") == 0);
    assert(HM_Get(&h, &data[5]) == 0);
    HM_Set(&h, &data[5], "one");
    //note: one empty cell left => cannot add more!
    HM_Destroy(&h);

    h = HM_Init(IntEqualFunc, IntHashFunc, 100000); //create larger table
    assert(HM_Get(&h, &data[5]) == 0);
    HM_Set(&h, &data[1], "newtable");
    HM_Set(&h, &data[1], "newtableX");
    assert(strcmp(HM_Get(&h, &data[1]), "newtableX") == 0);
}
```

Заметим, что здесь ключом является остаток от деления на 1000. В частности, целые числа 13, 1013, -987 считаются равными, т.к. они сравнимы по модулю 1000. Равенство подтверждается задаваемой функцией `EqualFunc`, и функция `HashFunc` выдаёт одинаковый хеш для всех этих ключей. При реализации хеш-таблицы вам не нужно думать об этом: надо лишь вызывать заданные функции для сравнения ключей.

Задача 7. Indirection

Источник:	повышенной сложности I
Имя входного файла:	`---`
Имя выходного файла:	`---`
Ограничение по времени:	5 секунд
Ограничение по памяти:	разумное

Код задачи: `4/7_resolve`

С учётом использования динамических библиотек получается, что вызов функции не всегда выполняется напрямую. Часто выполняется несколько косвенных переходов разного вида, перед тем, как процессор попадает на тело функции. В этой задаче предлагается научиться распознавать несколько видов таких косвенных переходов на архитектуре x86.

Первый тип — это безусловный переход в заданное место по относительному адресу:

```
000CEEEA E9 AD 35 00 00       jmp  _GetProcAddress@8 (0D249Ch)
```

В данном случае первый байт `E9` задаёт opcode команды, а остальные четыре байта — относительный адрес пункта назначения в little-endian. Относительный адрес (`000035AD`) получается вычитанием из абсолютного адреса пункта назначения (`000D249C`) адреса конца самой команды (`000CEEEA` + 5 байт). Относительный адрес является знаковым 32-битным целым, в частности, он может быть отрицательным.

Второй тип — это безусловный переход по абсолютному адресу, который записан в заданной ячейке памяти. Адрес ячейки задаётся абсолютный (важно: 32-битный режим):

```
000D249C FF 25 00 50 19 00    jmp  dword ptr [__imp__GetProcAddress@8 (0195000h)]
```

Здесь первые два байта `FF 25` кодируют opcode команды, а остальные четыре байта — откуда брать адрес пункта назначения. В данном случае из памяти по адресу `00195000` читается четырёхбайтовый адрес пункта назначения, и выполняется переход на него.

От вас требуется создать динамическую библиотеку с одной экспортированной функцией `resolve`. Сигнатура функции:

```
void* resolve(void* address);
```

В функцию передаётся абсолютный адрес в памяти, на который передаётся управление. Она должна вернуть адрес, в котором будет управление, когда закончатся переходы описанных выше типов. Если выполняемые косвенные переходы зацикливаются, то нужно вернуть `NULL` = 0.

Точное поведение функции должно быть таким:

1. Прочитать два байта по заданному адресу `address`.
2. Если первый байт имеет значение `E9`, то прочитать оставшиеся байты перехода первого типа и вычислить адрес `destAddr`, на который он ведёт.
3. Если прочитанные байты имеют значения `FF` и `25`, то прочитать ещё 4 байта перехода второго типа и загрузить адрес `destAddr`, на который он ведёт.
4. Если распознать переход не удалось, то вернуть `address` из функции.
5. Если распознать переход удалось, то вернуть `resolve(address)` (рекурсивный вызов).

Кроме того, от вас требуется реализовать защиту от бесконечной рекурсии: если такая ситуация обнаружена, нужно вернуть `NULL`.

Нужно отправить в систему тестирования ваш исходный код, а также скрипт `c.bat.txt`, который собирает из исходного кода динамическую библиотеку `resolve.dll` при помощи Visual C 32-bit. В системе тестирования код жюри будет импортировать вашу функцию и вызывать её для различных адресов. Убедитесь, что библиотека импорта `resolve.lib` также создаётся.

К сожалению, эта задача имеет смысл только в 32-битном режиме, из-за этого тестирования под Linux нет.

Гарантируется, что:

1. При правильном распознавании переходов ваш код сможет прочитать все нужные байты из памяти.
2. Общее количество выполненных косвенных переходов на протяжении всего тестирования не превысит $3 \cdot 10^5$; при условии, что ваша функция не будет выполнять никакой косвенный переход дважды в течение одного вызова `resolve`.

Состояние памяти может изменяться между вызовами функции `resolve`, но **не** изменяется в процессе работы `resolve`.

Пример использования функции `resolve` можно скачать здесь.

Задача 8. Хеш-таблица+

Источник:	повышенной сложности I
Имя входного файла:	`---`
Имя выходного файла:	`---`
Ограничение по времени:	разумное
Ограничение по памяти:	разумное

Код задачи: `4/8_hashmap2`

В этой задаче нужно реализовать "map" (отображение) с помощью хеш-таблицы. Задача продолжает предыдущую с некоторыми усложнениями.

Отличия этой задачи от предыдущей:

1. Пользователь **не** задаёт количество ячеек в хеш-таблице. Реализация должна сама следить за степенью заполненности таблицы и самостоятельно увеличивать таблицу при необходимости.
2. Нужно дополнительно реализовать операцию удаления заданного ключа из таблицы.
3. Функция сравнения и хеш-функция принимают контекст — указатель на произвольные дополнительные данные пользователя.
4. Функция `HM_Destroy` заменена на функцию `HM_Clear`, которая также освобождает память, но с таблицей после этого можно продолжать работать — таблица просто становится пустой.

Вы должны оформить вашу хеш-таблицу как динамическую библиотеку. В хедере `hashmap.h` должна быть объявлена структура `HashMap`, а также следующие типы и функции:

```
//pointer to key or value (untyped)
typedef const void *cpvoid;

//returns 1 if and only if two keys pointed by [a] and [b] are equal
//returns 0 otherwise
typedef int (*EqualFunc)(void *context, cpvoid a, cpvoid b);
//returns 32-bit hash of a key pointed by [key]
typedef uint32_t (*HashFunc)(void *context, cpvoid key);

//creates and returns new hash table with:
// [ef] -- function which compares keys for equality
// [hf] -- function which produces a hash for a key
// [context] -- a pointer which is passed to HashFunc and EqualFunc
HashMap HM_Init(EqualFunc ef, HashFunc hf, void *context);
//frees memory of hash map [self] and resets it to empty state
//it is allowed to use any operations on the table afterwards
void HM_Clear(HashMap *self);

//returns value corresponding to the specified key [key] in hash map [self]
//if [key] is not present in the map, then returns NULL
cpvoid HM_Get(const HashMap *self, cpvoid key);
//sets value [value] for the key [key] in hash map [self]
//if [self] already has some value for [key], it is overwritten
void HM_Set(HashMap *self, cpvoid key, cpvoid value);
//removes key [key] from the table [self]
//returns the value corresponding to key [key] before removal
//if key [key] is not present, then nothing is done, and NULL is returned
cpvoid HM_Remove(HashMap *self, cpvoid key);
```

Пример кода, который должен собираться вместе с библиотекой импорта и работать:

```
#include "hashmap.h"
#include <assert.h>
#include <string.h>

#define INTOF(ptr) (*(int*)(ptr))
int IntModEqualFunc(void *context, cpvoid a, cpvoid b) {
    return (INTOF(a) - INTOF(b)) % INTOF(context) == 0;
}
uint32_t IntModHashFunc(void *context, cpvoid key) {
    int t = INTOF(key) % INTOF(context);
    if (t < 0)
        t += INTOF(context);
    return t * 0xDEADBEEF;
}

int main() {
    int MOD = 1000;
    HashMap h = HM_Init(IntModEqualFunc, IntModHashFunc, &MOD);
    int data[] = {13, 174, 1013, -987, 0, 1};
    HM_Set(&h, &data[0], "hello");
    HM_Set(&h, &data[1], "world");
    assert(strcmp(HM_Get(&h, &data[2]), "hello") == 0);
    const char *old = HM_Remove(&h, &data[3]);
    assert(old && strcmp(old, "hello") == 0);
    assert(HM_Get(&h, &data[0]) == 0);
    old = HM_Remove(&h, &data[2]);
    assert(!old);
    HM_Set(&h, &data[4], "zero");
    assert(strcmp(HM_Get(&h, &data[4]), "zero") == 0);
    assert(strcmp(HM_Get(&h, &data[1]), "world") == 0);
    HM_Set(&h, &data[5], "one");
    assert(HM_Get(&h, &data[0]) == 0);
    HM_Clear(&h);   //note: frees memory
    assert(HM_Get(&h, &data[5]) == 0);
    assert(HM_Remove(&h, &data[3]) == 0);
    HM_Set(&h, &data[1], "newtable");
    HM_Set(&h, &data[1], "newtableX");
    assert(strcmp(HM_Get(&h, &data[1]), "newtableX") == 0);
}
```

Как и в примере к предыдущей задаче, здесь ключом является остаток от деления на 1000. Однако в этот раз модуль 1000 не зашит в код, а лежит в локальной переменной. Указатель на эту переменную передаётся как `context` при инициализации хеш-таблицы, которая затем передаёт его в функции сравнения и вычисления хеша.