Міністерство освіти і науки України

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

В. В. Шпурик, К. М. Олєнєва

ОСНОВИ ПРОГРАМУВАННЯ
Професійний підхід

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітньо-професійною програмою «Інженерія програмного забезпечення
інтелектуальних кібер-фізичних систем в енергетиці»
спеціальності 121 Інженерія програмного забезпечення

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського

2024

УДК 004.04
Ш83

Автори: *Шпурик Вадим Вадимович*, канд. техн. наук, доц.
Олєнєва Ксенія Миколаївна, старший викладач

Рецензент *Корнага Я. І.,* д-р техн. наук, доц., проф.
професор кафедри інформаційних систем та технологій,
декан факультету інформатики та обчислювальної техніки
Національного технічного університету України
«Київський політехнічний інститут ім. Ігоря Сікорського»

Відповідальний
редактор *Федорова Наталія Володимирівна*, д-р техн. наук., проф.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 2 від 08.11.2024 р.)
за поданням Вченої ради навчально-наукового інституту атомної та теплової енергетики
(протокол № 4 від 28.10.2024 р.)

Шпурик В. В.

Ш83 Навчальний посібник «Основи програмування. Професійний підхід» за освітньо-професійною програмою «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем в енергетиці» [Електронний ресурс] : навч. посіб. для здобувачів ступеня бакалавра за освіт. програмою «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем в енергетиці» спец. 121 Інженерія програмного забезпечення / В.В. Шпурик, К.М. Олєнєва ; КПІ ім. Ігоря Сікорського. – Електрон. текст. дані (1 файл). – Київ : КПІ ім. Ігоря Сікорського, 2024. – 106 с.

Навчальний посібник призначений для самостійного опрацювання студентами бакалаврату матеріалу дисципліни «Основи програмування. Частина 1. Базові конструкції». Матеріал навчального посібника має особливе значення для вивчення всього курсу програмування, оскільки у ньому продемонстровано покроковий процес створення програми, а також розглядаються основні бібліотеки для вирішення інженерних і наукових завдань, із якими студенти зустрічаються протягом усього часу вивчення дисципліни, а також у подальшій професійній діяльності

УДК 004.04

Реєстр. № П 24/25-109. Обсяг 3,95 авт. арк.
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Берестейський, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

Зміст

Передмова

Мова програмування С досягла поважного віку, їй вже скоро виповниться півстоліття. Всі ці роки мова С не стояла на місці, продовжувала активно розвиватись і у XXI столітті вона залишається ключовою мовою в енергетиці, машинобудуванні, авіації, космонавтиці та багатьох інших галузях. Світ працює на коді, написаному на мові С [2]. Вуличний світлофор, мікрохвильова піч на вашій кухні, комп'ютерна система вашого автомобіля, операційна система вашого смартфона або, напевно, будь-якого іншого пристрою, про який зазвичай навіть ніхто не замислюється, – все це містить програмні компоненти, написані мовою С.

Мета цього посібника – розглянути те, чого немає в більшості підручників і навчальних курсів по С, але обов’язково має бути в арсеналі розробника, і першим номером у цьому списку стоять бібліотеки. Програмний код існує не у вакуумі, і немає жодного сенсу створювати ще одну книгу, засновану на передумові, ніби знання синтаксису – це все, що необхідно для продуктивного використання мови програмування [6,7]. У всіх майже без винятку підручників з С є одна спільна особливість – вони досить докладно розбирають синтаксис і можливості стандартної бібліотеки і при цьому жодного слова не говорять про бібліотеки, які можуть стати в нагоді програмісту; ні слова про інструменти встановлення та екосистему, завдяки якій ці бібліотеки виявляються надійними та корисними [4].

На жаль у багатьох випадках однієї лише стандартної бібліотеки С може бути недостатньо [3]. Тому екосистема С вийшла за рамки стандарту, а це означає, що якщо ви збираєтеся вирішувати завдання, які будуть суттєво складніше вправ у підручнику, то вам необхідно знати, як викликати функції з поширених, але не описаних у стандарті ISO бібліотек [5]. Отже цей посібник починається там, де закінчується підручник з мови С – екосистема, що оточує мову.

Одна з сюжетних ліній – як користуватись бібліотекою для роботи зі зв'язаними списками, а не розробляти власні рішення з нуля. У цьому посібнику показано, як використовувати деякі можливості бібліотеки GLib для ефективного та елегантного керування даними у програмах на мові програмування С. Бібліотека GLib є результатом багаторічного вдосконалення та використовується багатьма програмами з відкритим кодом. GLib – це бібліотека утиліт загального призначення, яка надає структури/контейнери даних (функції та змінні, необхідні

для керування даними), яких бракує в стандартній бібліотеці мови С. Автор книги не ставив перед собою мету створити вичерпний опис, який би пояснював всі можливості бібліотеки GLib. Тема зв'язаних списків є тою стежкою, яка має привести вас до інших різноманітних можливостей бібліотеки. Використовуючи ту чи іншу функцію, розглянуту далі, необхідно також звертатися до офіційної документації GLib API для отримання додаткової детальної інформації. Слід зважити на те, що бібліотеки не є чимось застиглим, а постійно зазнають різного роду вдосконалення коду та розширення функціональності. Це одна з причин, яка не дозволяє зробити повне охоплення всіх можливостей бібліотеки та вимагає від розробника вивчення актуальної документації.

Багато прикладів, представлених у книзі, допоможуть вам підготуватися до того, з чим вам доведеться зіткнутися в реальних проектах. Технології не стоять на місці, інструменти швидко набувають і втрачають популярність, але фундаментальні підходи до вирішення проблем залишаються.

Ця книга не є введенням у мову програмування С. Для цього існує безліч книг, курсів та ресурсів з програмування. Представлені у книзі концепції зосереджені на суто практичних аспектах написання програм мовою С з використанням як стандартної, так і сторонніх бібліотек і зокрема GLib і наукової бібліотеки GSL. У посібнику даються в необхідному обсязі відомості щодо використання і побудови makefile.

Книга написана для програмістів Unix-подібних операційних систем, таких як Linux або macOS, чиї навички та досвід знаходяться на початковому та середньому рівні. Передбачається, що читач має базові знання С, набуті в процесі написання коду цією мовою.

Щоб отримати максимальну віддачу від цього посібника, читач повинен бути знайомим з UNIX-подібним середовищем і знати, як використовувати оболонку командного рядка. Читачеві також знадобляться базові інструменти програмування для компіляції прикладів вихідного коду, такі як компілятор на зразок GCC. Одна частина прикладів коду в цьому посібнику скомпільована за допомогою Clang, а інша – GCC. Читачеві також буде необхідно встановити бібліотеки GLib і GSL (runtime & development). Покликання цієї книги – допомога у роботі, тому читач може вільно використовувати наведений у цьому посібнику код у власних програмах та документації.

Навчальний посібник спонукає до подальшого вивчення програмування мовою С стандарту ANSI ISO і призначений для самостійного опрацювання студентами бакалаврату та спеціалітету матеріалу дисципліни «Основи

програмування. Частина 1», що входить до освітньо-професійної програми «*Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем в енергетиці*» спеціальності 121 *Інженерія програмного забезпечення*. Матеріал навчального посібника має особливе значення для вивчення всього курсу програмування, оскільки у ньому розглядаються основні бібліотеки для вирішення інженерних і наукових завдань, із якими студенти зустрічаються протягом усього часу вивчення дисципліни, а також у подальшій професійній діяльності.

При написанні цієї книги допомога ChatGPT та інших чат-ботів з генеративним штучним інтелектом не використовувалася.

Як організована ця книга

Книга складається з п’яти окремих глав. У перших трьох главах продемонстровано покроковий процес розв'язання завдань: аналіз, декомпозиція на окремі функції та перевірка працездатності створеного коду. В якості прикладів завдань взято алгоритм шифрування і дешифрування повідомлень за допомогою решітки Кардано, пошук решітки, яка використовувалася для отримання зашифрованого повідомлення, а також розпізнавання рукописних цифр за допомогою нейронної мережі (багатошарового перцептрона).

Якщо технічним прийомам і основам теорії можна навчити існуючими традиційними методами, то творчості можна навчити лише за допомогою зразків творчості, демонструючи і пояснюючи їх безпосередньо у процесі створення. У цьому – суть перших трьох глав цієї книги. Другу і третю глави рекомендується читати по черзі. Всі інші глави можна читати в довільному порядку. Ця книга написана з вірою в підхід до навчання «з нуля», натхненна цитатою Річарда Фейнмана «Того, що я не можу створити, я не розумію» (What I cannot create, I do not understand.).

Ця книга містить приклади вихідного коду. Вихідний код форматується `таким шрифтом фіксованої ширини`, щоб відокремити його від звичайного тексту.

У деяких випадках оригінальний вихідний код було переформатовано; додано розриви рядків і змінені відступи, щоб раціонально використати доступний простір сторінки в книзі. У рідкісних випадках довгі рядки коду чи команд оболонки містять маркери продовження рядків (➥). Крім того, коментарі у вихідному коді інколи видалялися з лістингів, коли код описано в тексті. У

фрагментах коду, що наводяться в тексті книги, іноді окремі рядки для скорочення обсягу пропущені і замінені трьома крапками ". . .".

Використання коду

Весь код, використаний у цій книзі, доступний у Git-репозиторії на GitHub. Посилання наведені у відповідних місцях в тексті книги. Якщо ви незнайомі з Git, то це система контролю версій, яка дозволяє відстежувати файли, що становлять проект. Колекція файлів, що знаходяться під контролем Git, називається «репозиторій». GitHub – це хостинг, який надає сховище для Git-репозиторіїв та зручний веб-інтерфейс.

Домашня сторінка GitHub мого репозиторію надає кілька способів роботи з кодом:

• ви можете клонувати репозиторій, натиснувши кнопку *Fork* у правому верхньому кутку. Якщо у вас ще немає облікового запису GitHub, вам потрібно створити його. Після цього у вас буде власний репозиторій GitHub, який ви зможете використовувати для відстеження коду, написаного вами під час роботи над цією книгою. Потім можна клонувати репозиторій, що означає, що ви копіюєте файли на свій комп'ютер;

• ви можете клонувати мій репозиторій без створення відгалуження; тобто можна зробити копію мого репозиторію на вашому комп'ютері. Для цього вам не потрібний обліковий запис GitHub, однак ви не зможете записати свої зміни назад до GitHub;

• якщо ви взагалі не хочете використовувати Git, то можете завантажити файли одним ZIP-архівом, скориставшись зеленою кнопкою з написом *Clone or download* (Клонувати або завантажити).

Глава 1. Решітка Кардано

1.1. Постановка задачі

Перш ніж поринути у таємничий світ бібліотек та інструментів мови С, давайте спробуємо уявити собі реальність, у якій ще майже нічого не створено: світ тільки народжується, а навколо лише безмежна і холодна пустеля, якою у ранкових багряних сутінках інколи промайне поодиноке програмне створіння на мові асемблера…

Все це більше схоже на сюжет для науково-фантастичного роману, тому будемо вважати, що в цій реальності ми непогано озброєні, бо в нашому розпорядженні є компілятор і стандартна бібліотека мови С, а також непереборне бажання впоратися із завданням, суть якого полягає в наступному. Необхідно розробити програму, яка реалізує алгоритм шифрування "Решітка Кардано".

Для ведення зашифрованого листування призначена спеціальна решітка – маска із заданими елементами, які використовуються для введення символів повідомлення, що шифрується або розшифровується. За допомогою решітки розміром 8x8 можна зашифрувати повідомлення максимальної довжини 64 символи. Алгоритм передбачає наявність однакових решіток у розпорядженні як відправника повідомлення, так і його одержувача. Детальний опис ідеї використання цього методу шифрування наведено у книзі Яківа Перельмана “Жива математика”, у розділі 6 “Математичні оповідання та головоломки”. Історія створення цього методу шифрування та загальна ідея описані в Cardan Grille. Друга частина цього завдання – зворотна задача, тобто відновлення втраченої решітки, спираючись на наявність зашифрованого та розшифрованого повідомлень.

1.2. Генерація решітки Кардано

Перше, що слід зробити, – створити функцію генерації решітки. Очевидно, що може існувати безліч варіантів решіток розміром 8x8. Кількість можливих варіантів перевищує 4 мільярди.

Для створення решітки необхідна базова матриця, яка має вигляд:

```
 1  2  3  4 13  9  5  1
 5  6  7  8 14 10  6  2
 9 10 11 12 15 11  7  3
13 14 15 16 16 12  8  4
 4  8 12 16 16 15 14 13
 3  7 11 15 12 11 10  9
 2  6 10 14  8  7  6  5
 1  5  9 13  4  3  2  1
```

У коді програми можна описати таку матрицю наступним чином:

```
const int matrix[][8] = {
      { 1,  2,  3,  4, 13,  9,  5,  1},
      { 5,  6,  7,  8, 14, 10,  6,  2},
      { 9, 10, 11, 12, 15, 11,  7,  3},
      {13, 14, 15, 16, 16, 12,  8,  4},
      { 4,  8, 12, 16, 16, 15, 14, 13},
      { 3,  7, 11, 15, 12, 11, 10,  9},
      { 2,  6, 10, 14,  8,  7,  6,  5},
      { 1,  5,  9, 13,  4,  3,  2,  1}
};
```

Якщо уважно придивитися до структури базової матриці, можна помітити певну закономірність – матриця складається з 4 окремих матриць, кожна з яких повернена за годинниковою стрілкою на фіксоване число градусів, починаючи з 90 градусів. Помічену особливість можна проігнорувати та ініціалізувати матрицю так, як було показано вище, але цього разу обійдемося без спрощувань і все-таки скористаємося знайденою особливістю. Функція для заповнення базової матриці:

```
const int SIDE = 8;

void init_quarters(int (*arr)[SIDE]) {
    const int dim = SIDE/2;
    int x = 1;
    for (int i = 0; i < dim; i++) {
        for (int j = 0; j < dim; j++) {
            arr[i][j] = x;                  // 1 quarter
            arr[j][SIDE-1-i] = x;           // 2 quarter
            arr[SIDE-1-j][i] = x;           // 3 quarter
```

```
            arr[SIDE-1-i][SIDE-1-j] = x;   // 4 quarter
            x++;
        }
    }
}
```

Виведення на екран вмісту матриці буде корисним для візуальної перевірки правильності роботи функції заповнення базової матриці:

```
void print_matrix(const int (*arr)[SIDE]) {
    for (int i = 0; i < SIDE; i++) {
        for (int j = 0; j < SIDE; j++) {
             printf("%2d ", arr[i][j]);
        }
        putchar('\n');
    }
}
```

Щоб упевнитися у правильності роботи функції заповнення базової матриці, додаємо в тіло функції `main` наступний код:

```
int matrix[SIDE][SIDE] = {0};
init_quarters(matrix);
print_matrix(matrix);
```

В результаті виконання цього коду на екрані з'явиться вже знайома матриця:

```
 1  2  3  4 13  9  5  1
 5  6  7  8 14 10  6  2
 9 10 11 12 15 11  7  3
13 14 15 16 16 12  8  4
 4  8 12 16 16 15 14 13
 3  7 11 15 12 11 10  9
 2  6 10 14  8  7  6  5
 1  5  9 13  4  3  2  1
```

Базову матрицю отримано і тепер можна створити функцію генерації решітки.

Як було зазначено вище, існує понад 4 мільярди можливих варіантів таких решіток. Для створення решітки доречно використовувати генератор випадкових чисел.

Функція генерації решітки може бути реалізована наступним чином:

```
const int QUARTER = SIDE * 2;

void gen_rand_cells(const int (*arr)[SIDE], int (*cells)[2],
                    const int length, const int lower, const int upper) {
      const int *ptr = &arr[0][0];
      int uniq[QUARTER] = {0};
      int count = 0;
      time_t t;
      srand((unsigned) time(&t));
      while (count < length) {
            int rand_idx = (rand() % (upper - lower + 1)) + lower;
            bool found = false;
            for (int i = 0; i < length; i++) {
                 if (uniq[i] == *(ptr + rand_idx)) {
                     found = true;
                     break;
                 }
            }
            if (!found) {
                uniq[count] = *(ptr + rand_idx);
                div_t output = div(rand_idx, SIDE);
                cells[count][0] = output.quot;
                cells[count][1] = output.rem;
                count++;
            }
      }
}
```

Зверніть увагу, що крім

```
#include <stdio.h>
```

потрібно також додати заголовки:

```
#include <time.h>
#include <stdlib.h>
#include <stdbool.h>
```

Додаткова функція, яка виводить на екран результати роботи генератора решіток, буде корисною для візуалізації роботи процесу генерації решітки:

```
void print_cells(const int (*arr)[SIDE], const int (*cell)[2]) {
    for (int i = 0; i < QUARTER; i++) {
        int row = cell[i][0];
        int col = cell[i][1];
        printf("matrix[%d][%d] = %2d\n", row, col, arr[row][col]);
   }
}
```

Додаємо в тіло функції `main` виклик функції генерації решіток:

```
int matrix[SIDE][SIDE] = {0};
int cells[QUARTER][2] = {0};

init_quarters(matrix);
print_matrix(matrix);

gen_rand_cells(matrix, cells, QUARTER, 0, SIDE*SIDE-1);
print_cells(matrix, cells);
```

Після виконання цього коду на екрані отримаємо один з можливих варіантів решітки:

```
matrix[2][5] = 11
matrix[4][0] =  4
matrix[3][5] = 12
matrix[6][1] =  6
matrix[4][6] = 14
matrix[6][0] =  2
matrix[0][5] =  9
matrix[3][0] = 13
matrix[5][0] =  3
matrix[4][4] = 16
matrix[5][6] = 10
matrix[7][7] =  1
matrix[6][5] =  7
matrix[5][3] = 15
matrix[6][7] =  5
matrix[4][1] =  8
```

Результат роботи функції `print_matrix(matrix)` тут не показано. Такий варіант візуалізації решітки може здатися незручним, тому створимо функцію, яка покаже решітку у вигляді матриці, в якій віконця представлені символом '1':

```
void print_grid(const int (*cell)[2]) {
    int grid[SIDE][SIDE] = {0};
    for (int i = 0; i < QUARTER; i++) {
        int row = cell[i][0];
        int col = cell[i][1];
        grid[row][col] = 1;
    }
    print_matrix(grid);
}
```

Результат виконання цієї функції:

```
 0  0  0  0  0  1  0  0
 0  0  0  0  0  0  0  0
 0  0  0  0  0  1  0  0
 1  0  0  0  0  1  0  0
 1  1  0  0  1  0  1  0
 1  0  0  1  0  0  1  0
 1  1  0  0  0  1  0  1
 0  0  0  0  0  0  0  1
```

Слід взяти до уваги те, що в коді використовується генератор випадкових чисел, тому при кожному запуску програми будуть створюватися нові решітки.

Така особливість створює певну проблему для подальшої розробки: потрібен один зафіксований результат виконання функції генерації решіток. Це можна зробити за допомогою спеціальної функції, яка створить решітку у вигляді вихідного коду мовою C. Згенерований код решітки можна включити у вихідний код і тимчасово використовувати для подальшої розробки:

```
void int_cells(const int (*arr)[2]) {
    printf("const int cells[][2] = {\n");
    for (int i = 0; i < QUARTER; i++) {
        printf("\t{%2d,%2d}", arr[i][0], arr[i][1]);
        if (i < QUARTER-1) {
            putchar(',');
        }
        putchar('\n');
```

```
    }
    printf("};\n");
}
```

Результат виконання цієї функції виглядає так:

```
const int cells[][2] = {
      { 2, 5},
      { 4, 0},
      { 3, 5},
      { 6, 1},
      { 4, 6},
      { 6, 0},
      { 0, 5},
      { 3, 0},
      { 5, 0},
      { 4, 4},
      { 5, 6},
      { 7, 7},
      { 6, 5},
      { 5, 3},
      { 6, 7},
      { 4, 1}
};
```

Перша частина завдання на цьому закінчена.

1.3. Шифрування та дешифрування за допомогою решітки Кардано

Тепер необхідно створити функції шифрування та дешифрування повідомлення за допомогою решітки. Для цього скористаємося наступним варіантом решітки:

```
const int cells[][2] = {
      { 5, 6},
      { 0, 5},
      { 2, 3},
      { 0, 6},
      { 4, 3},
```

```
        { 3, 2},
        { 7, 0},
        { 1, 7},
        { 3, 6},
        { 2, 2},
        { 2, 6},
        { 4, 6},
        { 7, 3},
        { 0, 3},
        { 7, 5},
        { 6, 1}
};
```

Повідомлення, яке потрібно зашифрувати чи розшифрувати, можна передати програмі різними способами. Наприклад, прочитати з файлу або передати через аргумент командного рядка. В даному випадку це не принципово, тому для спрощення завдання представимо повідомлення у вигляді рядка символів у вихідному коді:

```
const char *message =
       "Hey dude, do not let me down. Take a bad code and make it better.";
```

Згідно з описом алгоритму, тут знадобиться функція повороту решітки (матриці) за годинниковою стрілкою на 90 градусів.

Функція повороту матриці може бути реалізована так:

```
void rotate_clockwise(int (*arr)[SIDE]) {
   for (int i = 0; i < SIDE; i++){
      for (int j = 0; j < SIDE-i; j++) {
         int x = arr[i][j];
         arr[i][j] = arr[SIDE-1-j][SIDE-1-i];
         arr[SIDE-1-j][SIDE-1-i] = x;
      }
   }
   for (int i = 0; i < SIDE/2; i++) {
      for (int j = 0; j < SIDE; j++) {
         int x = arr[i][j];
         arr[i][j] = arr[SIDE-1-i][j];
         arr[SIDE-1-i][j] = x;
      }
   }
}
```

Крім того, потрібна функція запису символів повідомлення в вікна решітки. Реалізація такої функції має передбачати ситуацію, коли довжина повідомлення буде меншою за 64 символи. Іншими словами, якщо довжина повідомлення коротше 64 символів, то в цьому випадку залишаються вільні вікна. Якщо ці вікна залишити порожніми, це значно спростить процес злому шифру. Тому необхідно передбачити можливість заповнення вікон, що залишаються порожніми, випадковими символами.

Це ще не все. У повідомленні можуть бути пробіли між словами, розділові знаки і великі літери. Якщо залишити їх у зашифрованому повідомленні, це теж знизить стійкість зашифрованого повідомлення до злому. Отже, необхідно виключити із зашифрованого повідомлення такі символи. Великі літери слід замінити малими.

Функція заповнення решітки може бути реалізована так:

```
void fill_side(const char *message, int *count, int (*grid)[SIDE],
               char (*encrypted)[SIDE]) {
    srand(time(0));
    const char *ptr = &message[0];
    const size_t len = strlen(ptr);
    printf("[%2zu] \"%s\"\n", strlen(ptr + *count), ptr + *count);
    for (int i = 0; i < SIDE; i++) {
        for (int j = 0; j < SIDE; j++) {
            if (grid[i][j] == 1) {
               while ( (ispunct(message[*count])
                      || isspace(message[*count])) && *count < len) {
                   (*count)++;
               }
               if (*count < len) {
                  encrypted[i][j] = tolower(message[*count]);
               } else {
                  char lower = 'a', upper = 'z';
                  encrypted[i][j] = (rand() % (upper - lower + 1)) + lower;
               }
               (*count)++;
            }
        }
    }
}
```

Для контролю результатів роботи цієї функції скористаємось додатковою функцією виведення:

```
void print_symb(const char (*arr)[SIDE]) {
    for (int i = 0; i < SIDE; i++) {
      for (int j = 0; j < SIDE; j++) {
          printf("%2c ", arr[i][j] ? arr[i][j] : '.');
      }
      putchar('\n');
    }
}
```

Функція заповнення решітки має бути викликана 4 рази за кількістю сторін решітки всередині функції шифрування. Заповнюємо та повертаємо решітку 4 рази:

```
void encryptor(const char *message, char (*encrypted)[SIDE]) {
      int grid[SIDE][SIDE] = {0};
      init_grid(cells, grid);
      //print_matrix(grid);
      int count = 0; // counter of symbols in message
      for (int side = 0; side < 4; side++) {
          fill_side(message, &count, grid, encrypted);
          rotate_clockwise(grid);
          //print_symb(encrypted);
      }
}
```

Для перевірки роботи функції шифрування додамо до тіла функції `main` наступні рядки:

```
char encrypted[SIDE][SIDE] = {0};
const char *message =
          "Hey dude, do not let me down. Take a bad code and make it better.";
encryptor(message, encrypted);
```

В результаті виконання отримаємо на екрані:

```
[65] "Hey dude, do not let me down. Take a bad code and make it better."
 .  .  .  h  .  e  y  .
 .  .  .  .  .  .  .  d
```

```
 .  .  u  d  .  .  e  .
 .  .  d  .  .  .  o  .
 .  .  .  n  .  .  o  .
 .  .  .  .  .  .  t  .
 .  l  .  .  .  .  .  .
 e  .  .  t  .  m  .  .
[43] "e down. Take a bad code and make it better."
 e  .  .  h  .  e  y  .
 .  d  .  .  .  .  .  d
 .  .  u  d  o  w  e  .
 n  .  d  t  .  a  o  k
 .  .  .  n  .  .  o  .
 e  .  .  .  .  .  t  a
 .  l  b  a  d  c  .  o
 e  .  .  t  .  m  d  .
[21] "e and make it better."
 e  .  e  h  a  e  y  n
 .  d  .  .  .  .  d  d
 .  m  u  d  o  w  e  .
 n  a  d  t  k  a  o  k
 .  e  .  n  .  i  o  .
 e  t  .  .  b  e  t  a
 t  l  b  a  d  c  .  o
 e  t  e  t  r  m  d  .
[ 1] "."
 e  r  e  h  a  e  y  n
 m  d  h  s  p  u  d  d
 l  m  u  d  o  w  e  n
 n  a  d  t  k  a  o  k
 y  e  c  n  c  i  o  v
 e  t  a  p  b  e  t  a
 t  l  b  a  d  c  r  o
 e  t  e  t  r  m  d  j
```

Останній етап – дешифрування повідомлення. Дешифрування значно простіше, ніж процес шифрування і здійснюється за допомогою функції:

```
void decryptor(char *readme, const char (*encrypted)[SIDE]) {
      int grid[SIDE][SIDE] = {0};
      int count = 0;
      init_grid(cells, grid);
      //print_matrix(grid);
      for (int side = 1; side <= 4; side++) {
            printf("Side #%d : ", side);
```

```
            for (int i = 0; i < SIDE; i++) {
                  for (int j = 0; j < SIDE; j++) {
                        if (grid[i][j] == 1) {
                              putchar(encrypted[i][j]);
                              readme[count] = encrypted[i][j];
                              count++;
                        }
                  }
            }
            putchar('\n');
            rotate_clockwise(grid);
      }
}
```

При дешифруванні використовується та сама решітка, що і при шифруванні!

Зберемо все разом. Додамо до тіла функції `main` рядки:

```
char readme[SIDE*SIDE+1] = {0};
decryptor(readme, encrypted);
printf("readme=%s\n", readme);
```

Тут `readme` – символьний масив для збереження розшифрованого повідомлення.

Після виконання функції дешифрування отримаємо:

```
Side #1 : heydudedonotletm
Side #2 : edowntakeabadcod
Side #3 : eandmakeitbetter
Side #4 : ovsgpgtjmfcarytz
readme=heydudedonotletmedowntakeabadcodeandmakeitbetterovsgpgtjmfcarytz
```

Зверніть увагу на те, що з міркувань збільшення криптостійкості, описаних вище, вихідне повідомлення доповнено в кінці випадковими символами “`ovsgpgtjmfcarytz`”. Через те, що розділові знаки та пробіли прибрані, а також великі літери замінені малими, вихідний вигляд повідомлення змінився, але саме повідомлення цілком читаємо.

Візьмемо в якості теста повідомлення, довжина якого з урахуванням видалення розділових знаків і пропусків перевищує допустимі 64 символи:

```
const char *message =
   "Hey dude, don't let me down. Take a bad soft and make it better. "
   "Remember to let it under your skin. Then you'll begin to make it better.";
```

Результат виконання програми:

```
[137] "Hey dude, don't let me down. Take a bad soft and make it better. Remember
to let it under your skin. Then you'll begin to make it better."
 .  .  .  h  .  e  y  .
 .  .  .  .  .  .  .  d
 .  .  u  d  .  .  e  .
 .  .  d  .  .  .  o  .
 .  .  .  n  .  .  t  .
 .  .  .  .  .  .  l  .
 .  e  .  .  .  .  .  .
 t  .  .  m  .  e  .  .
[115] " down. Take a bad soft and make it better. Remember to let it under your
skin. Then you'll begin to make it better."
 d  .  .  h  .  e  y  .
 .  o  .  .  .  .  .  d
 .  .  u  d  w  n  e  .
 t  .  d  a  .  k  o  e
 .  .  .  n  .  .  t  .
 a  .  .  .  .  .  l  b
 .  e  a  d  s  o  .  f
 t  .  .  m  .  e  t  .
[93] " and make it better. Remember to let it under your skin. Then you'll begin
to make it better."
 d  .  a  h  n  e  y  d
 .  o  .  .  .  .  m  d
 .  a  u  d  w  n  e  .
 t  k  d  a  e  k  o  e
 .  i  .  n  .  t  t  .
 a  b  .  .  e  t  l  b
 t  e  a  d  s  o  .  f
 t  e  r  m  r  e  t  .
[71] "emember to let it under your skin. Then you'll begin to make it better."
 d  e  a  h  n  e  y  d
 m  o  e  m  b  e  m  d
 r  a  u  d  w  n  e  t
```

```
 t  k  d  a  e  k  o  e
 o  i  l  n  e  t  t  t
 a  b  i  t  e  t  l  b
 t  e  a  d  s  o  u  f
 t  e  r  m  r  e  t  n

Side #1 : heydudedontletme
Side #2 : downtakeabadsoft
Side #3 : andmakeitbetterr
Side #4 : emembertoletitun
readme=heydudedontletmedowntakeabadsoftandmakeitbetterremembertoletitun
```

Очевидно, що вихідне повідомлення не помістилося повністю у відведені позиції решітки. Це цілком очікуваний результат.

1.4. Відновлення решітки

Перейдемо тепер до зворотного завдання, яке нагадує злом шифру. Вважаємо, що нам до рук потрапив текст зашифрованого та розшифрованого повідомлень. Для зручності подальшого розгляду зашифрований текст подаємо у вигляді двовимірного масиву:

```
const char encrypted[][8] = {
      {'e' , 'a' , 'e' , 'n' , 'd' , 'u' , 'f' , 'k' },
      {'m' , 'a' , 'd' , 'k' , 'e' , 'i' , 'q' , 'e' },
      {'t' , 'o' , 'w' , 'n' , 'b' , 'e' , 't' , 't' },
      {'t' , 'a' , 'k' , 'e' , 'k' , 'r' , 'b' , 'r' },
      {'e' , 'a' , 'h' , 'b' , 'e' , 'x' , 'g' , 'u' },
      {'y' , 'd' , 'u' , 'd' , 'f' , 'm' , 'c' , 'e' },
      {'a' , 'd' , 'd' , 'o' , 'n' , 'g' , 'o' , 't' },
      {'c' , 'o' , 'd' , 'l' , 'e' , 'r' , 't' , 'm' }
};
```

Розшифрований текст подаємо у вигляді фрагмента повідомлення:

```
const char *message = "heydudedonotletm";
```

Наше завдання полягає тепер у пошуку решітки, яка використовувалася для отримання зашифрованого повідомлення.

Ми не будемо розглядати будь-які витончені схеми злому, а спробуємо написати розв'язання задачі методом "грубої сили", тобто спираючись більшою мірою на обчислювальну міць сучасного комп'ютера.

Для вирішення цього завдання немає потреби обертати матрицю. Достатньо мати тільки текст, отриманий за допомогою першої сторони решітки, щоб прийняти рішення, що шифр зламаний. Завдання зводиться до послідовної генерації варіантів решіток. Кожну згенеровану решітку слід “прикладати” до зашифрованого повідомлення, читати символи та порівнювати їх із фрагментом розшифрованого повідомлення.

Основна складність полягає у послідовному переборі варіантів решіток. Відомо, що решітка – це 16 унікальних “віконець” у базовій матриці. Базову матрицю було розглянуто у першій частині завдання:

```
 1  2  3  4 13  9  5  1
 5  6  7  8 14 10  6  2
 9 10 11 12 15 11  7  3
13 14 15 16 16 12  8  4
 4  8 12 16 16 15 14 13
 3  7 11 15 12 11 10  9
 2  6 10 14  8  7  6  5
 1  5  9 13  4  3  2  1
```

Нагадую, базова матриця складається з 4-х субматриць, отриманих шляхом послідовного повороту матриці 4x4 за годинниковою стрілкою. Якщо розглядати кожну з 4 матриць як 16-розрядне ціле число без знака і використовувати його розряди для того, щоб забезпечити унікальність віконців у кожній решітці, то тоді можна використати бітову арифметику для перевірки унікальності віконців у кожній із згенерованих решіток.

Іншими словами, для вирішення цього завдання найбільш очевидним способом, знадобиться чотири цикли (за кількістю матриць 4x4):

```
typedef unsigned int UINT;
const int SIDE = 8;

const UINT top = 65536; // 65536 = 2^(SIDE*2)

for (UINT i = 0; i < top; i++) {
. . .
```

```
    for (UINT j = 0; j < top; j++) {
. . .
        for (UINT k = 0; k < top; k++) {
. . .
            for (UINT m = 0; m < top; m++) {
. . .
```

Крім того, цікавить час, витрачений програмою на пошук відповідних решіток. Для цього помістимо код пошуку решітки всередину наступного фрагмента коду:

```
#include <time.h>

const clock_t before = clock();

. . . // код пошуку варіанта решітки

const clock_t difference = clock() - before;
const int msec = difference * 1000 / CLOCKS_PER_SEC;
printf(":( Time taken %d seconds %d milliseconds\n", msec/1000, msec%1000);
```

Спираючись на всі вищеописані міркування, отримаємо такий код:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

const int matrix[][SIDE] = {
    { 1,  2,  3,  4, 13,  9,  5,  1},
    { 5,  6,  7,  8, 14, 10,  6,  2},
    { 9, 10, 11, 12, 15, 11,  7,  3},
    {13, 14, 15, 16, 16, 12,  8,  4},
    { 4,  8, 12, 16, 16, 15, 14, 13},
    { 3,  7, 11, 15, 12, 11, 10,  9},
    { 2,  6, 10, 14,  8,  7,  6,  5},
    { 1,  5,  9, 13,  4,  3,  2,  1}
};

const clock_t before = clock();
const UINT top = 65536; // 65536 = 2^(SIDE*2)
int grid[SIDE][SIDE] = {0};
```

```
const size_t area = sizeof(grid[0][0]) * SIDE * SIDE;
unsigned long int iter = 0;

...
for (UINT i = 0; i < top; i++) {
  UINT uniq_i = i;
  for (UINT j = 0; j < top; j++) {
    if (uniq_i & j)
      continue;
    UINT uniq_j = uniq_i | j;
    for (UINT k = 0; k < top; k++) {
      if (uniq_j & k)
        continue;
      UINT uniq_k = uniq_j | k;
      int base[SIDE][SIDE] = {0};
      make_grid_quarter(base, i, 1);
      make_grid_quarter(base, j, 2);
      make_grid_quarter(base, k, 3);
      for (UINT m = 0; m < top; m++) {
        if ((uniq_k ^ m) == 65535) {
          printf("%5d %5d %5d %5d\n", i, j, k, m);
          memcpy(grid, base, area);
          make_grid_quarter(grid, m, 4);
          char readme[DIGITS + 1] = {'\0'};
          decryptor(grid, encrypted, readme);
          if (strcmp(readme, message) == 0) {
            printf("Bingo!\n");
            printf("%5d %5d %5d %5d\n", i, j, k, m);
            print_grid(matrix, grid);
            const clock_t difference = clock() - before;
            const int msec = difference * 1000 / CLOCKS_PER_SEC;
            printf("Time taken %d seconds %d milliseconds\n", msec / 1000,
                   msec % 1000);
            return 0;
          }
          iter++;
        }
      }
    }
  }
}
...
```

У коді використано деякі додаткові функції. Код цих функцій наведено нижче:

```
const int DIGITS = SIDE * 2;

void int_to_bin_digit(UINT in, int count, int *out) {
    UINT mask = 1U << (count-1);
    for (int i = 0; i < count; i++) {
        out[i] = (in & mask) ? 1 : 0;
        in <<= 1;
    }
}

void get_coords(const int quarter, const int index, int *x, int *y) {
      const div_t t = div(index, SIDE/2);
      const int row = t.quot;
      const int col = t.rem;

      switch (quarter) {
          case 1:
            *x = row;
            *y = col;
            break;
          case 2:
            *x = col;
            *y = SIDE-1-row;
            break;
          case 3:
            *x = SIDE-1-col;
            *y = row;
            break;
          case 4:
            *x = SIDE-1-row;
            *y = SIDE-1-col;
            break;
      }
}

void make_grid_quarter(int (*grid)[SIDE], const UINT value, const int quarter) {
      int digit[DIGITS];
      int_to_bin_digit(value, DIGITS, digit);
      for (int i = 0; i < DIGITS; i++) {
            if (digit[i] == 0) {
                  continue;
```

```
            }
            int x, y;
            get_coords(quarter, i, &x, &y);
            grid[x][y] = 1;
        }
}
```

Результат виконання цієї програми для заданих умов завдання:

```
Bingo!
    0     0  8830 56705
 .  .  .  .  .  .  .  .
 .  .  .  .  .  .  .  .
 .  .  .  .  .  .  .  .
 .  .  .  .  .  .  .  .
 .  . 12  . 16  .  .  .
 3  7 11 15  .  .  .  9
 .  . 10 14  8  .  6  5
 .  .  . 13  4  .  2  1

Time taken 0 seconds 917 milliseconds
```

Злом такого шифру не слід вважати тривіальним завданням. Візьміть до уваги те, що тут продемонстровано прямолінійний варіант вирішення завдання – злом методом грубої сили. Метод послідовного перебору варіантів решіток має ряд недоліків, які нескладно помітити при спробі задіяти запропоноване вище рішення для визначення решітки, яка була використана в прикладі першої частини завдання. Ви також мабуть звернете увагу на те, що час вирішення завдання може бути значно більшим, ніж показано у наведеному вище прикладі. Подібний підхід, що спирається на метод грубої сили, зовсім не виключає необхідності створювати більш витончений код, який виконуватиметься швидше. Тут є достатній простір для творчості, який залишається вам. Моє завдання полягало в тому, щоб пройти разом з вами весь шлях від постановки задачі, її аналізу та пошуку відповідного варіанта рішення до отримання кінцевого результату, тим самим показавши вам на конкретному прикладі можливий підхід до вирішення подібних завдань.

Вихідний код рішення цього завдання – це наступні чотири файли:

```
$ ls -l
```

```
total 56
-rw-r--r-- 1 vadim  staff  5058 31 лип 13:38 brute.c
-rw-r--r-- 1 vadim  staff  4152 31 лип 13:38 encrypt.c
-rw-r--r-- 1 vadim  staff  2854 31 лип 13:39 grid.c
-rw-r--r-- 1 vadim  staff  4486 31 лип 13:39 solver.c
```

У цій главі продемонстровано покроковий процес розв'язання класичної задачі використання решітки Кардано для шифрування і дешифрування коротких повідомлень, а також відновлення втраченої решітки методом грубої сили. На прикладі розв'язання цієї задачі показані базові етапи розробки програмного забезпечення: аналіз, декомпозиція на окремі функції та перевірка працездатності створеного коду. Від вас було потрібно, як кажуть у цьому випадку фокусники, стежити за руками, тобто стежити за тим, як на ваших очах народжується код, розмірковуючи, сумніваючись, відчуваючи осяяння власними ідеями.

Тепер саме час завантажити на свій комп'ютер вихідний код за адресою на GitHub: https://github.com/iamvadimov/grille та почати з ним роботу. Код свідомо було написано так, щоб залишити достатній простір для вашої самостійної творчості. Особливо це стосується відновлення втраченої решітки.

Глава 2. На початку була матриця

2.1. Множення матриць

Матрицею розміру n×m називається прямокутна таблиця спеціального вигляду, що складається з n рядків та m стовпців, заповнених числами. Елементи матриці **A** позначають a_{ij}, де i – номер рядка, в якому знаходяться елементи матриці, j – номер стовпця. Кількість рядків та стовпців задають розміри матриці. Зазвичай матрицю програмісти уявляють у вигляді двовимірного масиву.

Як знайти добуток матриць? Для обрахунку добутку матриць, зробимо детальний розв'язок прикладу, який дозволить зрозуміти алгоритм розв'язання таких задач.

Результатом множення матриці $\mathbf{A}_{m\times n}$ на матрицю $\mathbf{B}_{n\times k}$ буде матриця $\mathbf{C}_{m\times k}$ така, що елемент матриці **C**, який знаходиться в i-тому рядку та j-тому стовпчику (c_{ij}), дорівнює сумі добутків елементів i-того рядку матриці **A** на відповідні елементи j-того стовпчика матриці **B**. Дві матриці можна перемножити якщо кількість стовпчиків першої матриці дорівнює кількості рядків другої матриці.

Трохи спростим приклад і у якості матриці **B** візьмемо матрицю, яка містить тільки один стовпчик. Таку матрицю називають матрицею-стовпцем, або вектором. Наприклад:

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B} = \begin{pmatrix} 0.9 & 0.3 & 0.4 \\ 0.2 & 0.8 & 0.2 \\ 0.1 & 0.5 & 0.6 \\ 0.9 & 0.7 & 0.9 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 2.9 \\ 1.6 \\ 1.9 \\ 4.3 \end{pmatrix}$$

Компоненти матриці **C** обраховуються наступним чином:

$$c_{11} = a_{11} \cdot b_{11} + a_{12} \cdot b_{21} + a_{13} \cdot b_{31} = 0.9 \cdot 2 + 0.3 \cdot 1 + 0.4 \cdot 2 = 1.8 + 0.3 + 0.8 = 2.9$$

$$c_{21} = a_{21} \cdot b_{11} + a_{22} \cdot b_{21} + a_{23} \cdot b_{31} = 0.2 \cdot 2 + 0.8 \cdot 1 + 0.2 \cdot 2 = 0.4 + 0.8 + 0.4 = 1.6$$

$$c_{31} = a_{31} \cdot b_{11} + a_{32} \cdot b_{21} + a_{33} \cdot b_{31} = 0.1 \cdot 2 + 0.5 \cdot 1 + 0.6 \cdot 2 = 0.2 + 0.5 + 1.2 = 1.9$$

$$c_{41} = a_{41} \cdot b_{11} + a_{42} \cdot b_{21} + a_{43} \cdot b_{31} = 0.9 \cdot 2 + 0.7 \cdot 1 + 0.9 \cdot 2 = 1.8 + 0.7 + 1.8 = 4.3$$

Програмна реалізація алгоритму обрахунку добутку матриці і вектора:

```
/* File mxv.c */
#include <stdio.h>

const int ROWS = 4;
const int COLS = 3;

void mvp(int rows, int cols, double *matrix, double *vector, double *resvec) {
  /* Matrix-vector product */
  for (int i = 0; i < cols; i++) {
    for (int j = 0; j < rows; j++) {
      resvec[j] += *((matrix + j * cols) + i) * vector[i];
    }
  }
}

void printv(double *vec, int len) {
  for (int i = 0; i < len; i++) {
    printf("%g\t", vec[i]);
  }
  puts("");
}

int main() {
  double matrix[ROWS][COLS] = {
      {0.9, 0.3, 0.4},
      {0.2, 0.8, 0.2},
      {0.1, 0.5, 0.6},
      {0.9, 0.7, 0.9},
  };
  double vector[COLS] = {2, 1, 2};
  double result_vector[ROWS] = {0};
  mvp(ROWS, COLS, (double *)matrix, vector, result_vector);
  printv(result_vector, ROWS);
  return 0;
}
```

У разі, якщо оновлення результуючого вектора за допомогою арифметики вказівників у тілі функції `mvp` вам не до смаку, то можна реалізувати цю функцію інакше:

```
void mvp(int rows, int cols, double matrix[rows][cols], double vector[cols],
          double result_vector[rows]) {
  for (int i = 0; i < cols; i++) {
    for (int j = 0; j < rows; j++) {
      result_vector[j] += matrix[j][i] * vector[i];
    }
  }
}
```

і викликати її так

```
mvp(ROWS, COLS, matrix, vector, result_vector);
```

Функція `printv` роздруковує вміст вектора, довжина якого передається через її другий параметр.

Компіляція файлу з вихідним кодом і запуск на виконання дадуть нам очікуваний результат:

```
$ clang -Wall mxv.c
$ ./a.out
2.9     1.6     1.9     4.3
```

Вдоскональте цей код: спробуйте самостійно реалізувати перевірку можливості обрахунку добутку матриць – кількість стовпчиків першої матриці має дорівнювати кількості рядків другої матриці.

Примітка: для компіляції коду прикладів перших трьох глав цієї книги використовувався компілятор:

```
$ clang --version
Apple clang version 15.0.0 (clang-1500.3.9.4)
Target: x86_64-apple-darwin23.5.0
```

Є один нюанс, на який слід звернути увагу. Якщо замість компілятора clang ви використаєте gcc, то йому (в залежності від його версії) дещо в цьому коді може не сподобатися. Ви можете отримати від компілятора gcc, наприклад, версії 7.4.0 приблизно таке повідомлення:

```
variable-sized object may not be initialized
   double matrix[ROWS][COLS] = {
   ^~~~~~
```

Спробуйте самостійно скомпілювати цей файл за допомогою компілятора gcc і проаналізуйте всі отримані від нього повідомлення.

Щоб впоратися з цією особливістю компіляторів, слід замінити в коді константи :

```
const int ROWS = 3;
const int COLS = 4;
```

на макроси:

```
#define ROWS 3
#define COLS 4
```

і цей код стане “їстівним” для обох компіляторів.

2.2. Поширення сигналів в нейронній мережі

Використовуючи розпізнавання голосу у смартфоні або у Google Translate, ви неодмінно маєте справу з нейронною мережею, натренованою глибоким навчанням. За останні кілька років глибоке навчання забезпечило компанії Google прибуток, достатній для того, щоб покрити витрати на всі футуристичні проекти Google X, включаючи безпілотні автомобілі, окуляри Google Glass та науково-дослідний проект Google Brain. Google однією з перших почала застосовувати глибоке навчання. Ще у 2013 році Google найняла Джеффрі Хінтона, батька-засновника глибокого навчання, і зараз інші компанії намагаються її наздогнати. Ну і нам теж не до смаку пасти задніх, і якщо ви опанували обрахунок добутку матриць і розібралися з програмним кодом, що втілює цей алгоритм, то нам час перейти до вирішення нового завдання. Спробуємо використати ці знання для отримання відгуку нейронної мережі на заданий вхідний сигнал.

Коротко поясню ідею. Нейронна мережа – це математична функція, яка приймає вхідні та виробляє вихідні дані. Нейронна мережа складається з шарів, кожен із яких можна розглядати як ряд «нейронів». Кожен нейрон у нейронній мережі приймає вхідний сигнал від декількох нейронів, що знаходяться перед

ним, і, у свою чергу, також передає сигнал багатьом іншим нейронам у разі збудження. Одним із способів відтворення такої поведінки нейронів, що спостерігається в живій природі, у штучній моделі є створення багатошарових нейронних структур, в яких кожен нейрон з'єднаний з кожним з нейронів у попередньому та наступному шарах. Ця ідея пояснюється наступною ілюстрацією.

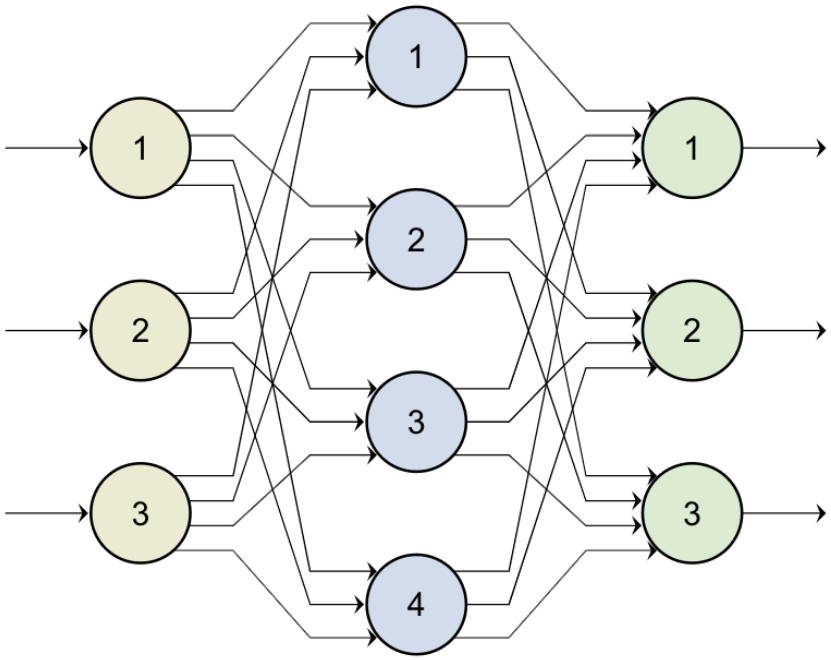


Рисунок 2.1 – Приклад нейронної мережі

Як ми можемо змоделювати штучний нейрон? Перші штучні нейронні мережі були названі перцептронами. Кожен нейрон має кілька входів (i). Нейрон, підсумовуючи відповідні вхідні значення, обчислює результуючу суму, яка стане аргументом функції сигмоїди (σ), що управляє вихідним значенням нейрона. Така схема відбиває принцип роботи нейронної мережі. Нижче наведена діаграма (Рис. 2.2) ілюструє ідею комбінування вхідних значень і обробку результуючої суми функцією сигмоїди. Якщо комбінований сигнал недостатньо сильний, сигмоїда пригнічує вихідний сигнал, але якщо сума вхідних сигналів досить велика, то функція σ збуджує нейрон.

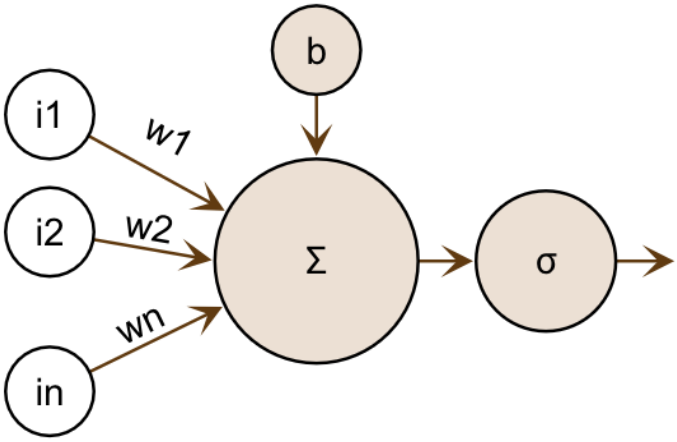


Рисунок 2.2 – Схема нейрона

Функція, яка отримує вхідний сигнал, та генерує вихідний сигнал з урахуванням порогового значення, називається функцією активації. З математичної точки зору існує безліч таких функцій активації, які могли б забезпечити такий ефект. В якості приклада функції активації було згадано S-подібну функцію, яку називають сигмоїдою σ, або сигмоїдальною функцією:

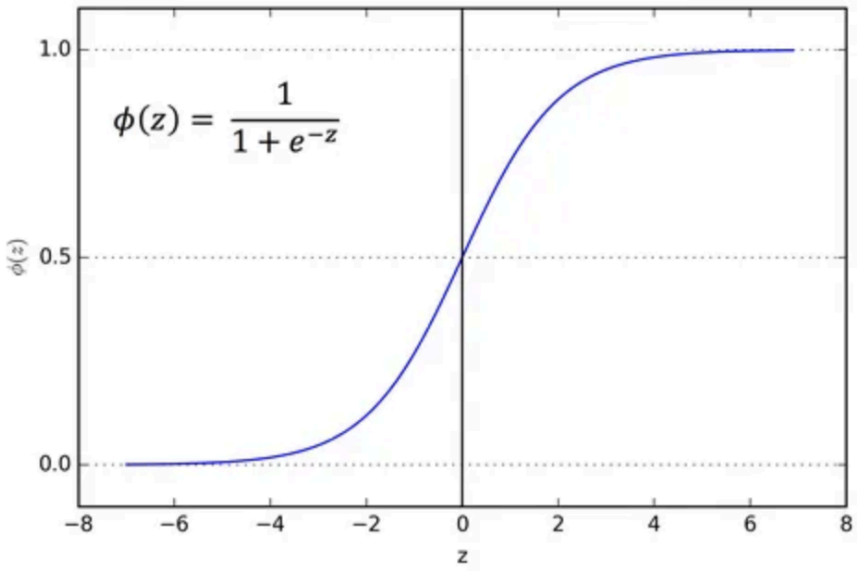


Рисунок 2.3 – Сигмоїдальна функція

Буквою e математиці прийнято позначати константу, рівну 2.71828. Це так зване трансцендентне число. Якщо ви раптом не знаєте, що це таке, можете знайти цей термін в інтернеті; його пояснення виходить за межі теми цієї книги.

Дослідниками в галузі штучного інтелекту використовуються також інші функції активації аналогічного виду, але сигмоїда проста і дуже популярна, тому

вона буде в нашому випадку підходящим вибором. У перцептроні вхідні дані (i) множаться на вагові коефіцієнти (w) і підсумовуються. Крім того, існує сутність під назвою зсув (bias, b) або параметр активації, який також додається до суми (Σ) вхідних даних, помножених на ваги. Параметр активації дає додаткову гнучкість навчальним можливостям персептрона, хоча перцептрон може працювати і без нього. Сума Σ проходить через сигмоїдальну функцію σ і перетворює Σ у вихідний сигнал нейрона.

Ми далі не заглиблюватимемося в теорію побудови нейронних мереж. На цю тему написано багато хороших книг, назву однієї з них ви знайдете в списку літератури [1].

Повернемося до наведеного вище зображення нейронної мережі (Рис. 2.1). На цій ілюстрації представлені три шари, перший і останній з яких включають по три штучні нейрони, або вузли, а середній шар – чотири вузли. Легко помітити, тут кожен вузол з'єднаний з кожним із вузлів попереднього та наступного шарів. З кожним з'єднанням асоціюється певна вага w. Низький ваговий коефіцієнт послаблює сигнал, високий – посилює його. Перший шар вузлів – вхідний. Його єдине призначення – представляти вхідні сигнали. У вхідних вузлах функція активації до вхідних сигналів не застосовується. Зараз немає необхідності висувати щодо цього якісь розумні доводи. Просто сприймаємо як даність: перший шар нейронних мереж є лише вхідним шаром, що представляє вхідні сигнали.

Розрахувати поширення вхідних сигналів по всіх шарах, поки вони не трансформуються у вихідні сигнали, насправді доволі просто, за умови, що ви розумієте, як працює множення матриць. Математики розробили надзвичайно компактний спосіб запису операцій, які доводиться виконувати для обчислення вихідного сигналу нейронної мережі, навіть якщо вона містить багато шарів та вузлів. Ця компактність сприятлива не тільки для людей, які виписують або читають відповідні формули, але й для комп'ютерів, оскільки програмні інструкції виходять коротшими і виконуються набагато швидше. У цьому компактному підході передбачається використання матриць, про які йшлося раніше.

Для початку нам знадобиться кілька матриць. Перша (вектор) містить сигнали вхідного шару. Друга містить вагові коефіцієнти для зв'язків між вузлами двох шарів – вхідного та другого (так званого прихованого шару). Результатом

множення цих двох матриць є об’єднаний згладжений сигнал, що надходить на вузли другого (вихідного) шару.

Обчислення вхідних сигналів, що надходять на кожен із вузлів другого шару, може бути виконано з використанням матричного множення. Немає потреби в тому, щоб якось особливо дбати про те, скільки вузлів входить у кожен шар. Збільшення кількості шарів призводить до збільшення розміру матриць. Щоб отримати вихідний сигнал другого шару, потрібно застосувати сигмоїду до кожного окремого елемента вектора, отриманого в результаті множення матриці вагових коефіцієнтів на вектор вхідних сигналів. Такий підхід застосовується для обчислення сигналів, що проходять від одного шару до наступного шару. Наприклад, за наявності трьох шарів ми просто знову виконаємо операцію множення матриць, використовуючи вихідні сигнали другого шару як вхідні для третього, але, зрозуміло, попередньо скомбінувавши їх і згладивши за допомогою додаткових вагових коефіцієнтів.

Задумайтесь на хвилину про значущість зв'язків між нейронами. У вашому мозку сотня мільярдів нейронів, але справжнє джерело вашої індивідуальності – зв'язок між цими нейронами. Кожен нейрон створює щонайменше тисячу зв'язків із іншими нейронами. Тобто у вашому мозку близько сотні трильйонів зв'язків. Наскільки велике це число? У Чумацькому Шляху приблизно 400 мільярдів зірок. У трильйоні – 1000 мільярдів. Виходить, що у вашому мозку більше нейронних зв'язків, ніж зірок у 5 тисячах Чумацьких Шляхів. Ось реальний масштаб індивідуальності – розмір вашої сутності. У світі не було, немає і мабуть ніколи не буде людини з ідентичною сотнею трильйонів зв'язків. Те, що ви запам'ятали, забули, що змушує вас сміятися, нервувати, радіти, що лякає, заспокоює чи виснажує – все це становить унікальну структуру, яка властива лише вам. Світ, який ви бачите протягом життя, видно тільки вам. Ваші реакції на цей світ – тільки ваші. Те, що ви любите – дії, морозиво в літню спеку, сміх коханої людини, рядок комп'ютерного коду, ідеальна відповідність візерунків двох потертих плиток на підлозі вашої кухні – все це доступно тільки вам. Ви все це пам'ятаєте. Усередині вас цілі галактики: вони світитимуть яскраво лише до тих пір, поки ви живі. Варто їм згаснути після вашої смерті, і ніщо і ніхто ніколи не сяятиме так само, як вони…

З теорією і філософією закінчено. Тепер час подивитися, як можна втілити теорію в коді, звернувшись до конкретного прикладу, який представлятиме

нейронну мережу з трьома шарами по три вузли у вхідному і вихідному шарі та чотири вузли у прихованому шарі. Така мережа була представлена раніше.

2.3. Використання матричного множення у мережі з трьома шарами

Відомо, перший шар – вхідний, проміжний шар називається прихованим шаром, а третій – вихідний. Тепер можна розпочати створення власної нейронної мережі за допомогою мови С.

Матриця ваг `Wih` (input-hidden) містить вагові коефіцієнти для зв'язків між вхідним та прихованим шарами. Коефіцієнти для зв'язків між прихованим та вихідним шарами будуть зберігатися в іншій матриці, яку позначено `Who` (hidden-output) прихований-вихідний.

Нижче представлені вхідний вектор і всі вагові коефіцієнти (кожен з них вибирався як випадкове число). Ніякі особливі міркування поки що за їх вибором не стояли.

```
const int INODES = 3;
const int HNODES = 4;
const int ONODES = 3;
double Wih[HNODES][INODES] = {
    {0.9, 0.3, 0.4},
    {0.2, 0.8, 0.2},
    {0.1, 0.5, 0.6},
    {0.9, 0.7, 0.9},
};
double Who[ONODES][HNODES] = {
    {0.3, 0.7, 0.5, 0.6},
    {0.6, 0.5, 0.2, 0.3},
    {0.8, 0.1, 0.9, 0.4},
};
double inputs[INODES] = {0.9, 0.1, 0.8};
```

Ваговий коефіцієнт для зв'язку між першим вхідним вузлом та першим вузлом проміжного прихованого шару $w_{11} = 0.9$. Так само ваговий коефіцієнт для зв'язку між другим вхідним вузлом і другим вузлом прихованого шару $w_{22} = 0.8$. Між першим і другим $w_{12} = 0.2$.

```
/* File simple_query.c */
```

```c
#include <math.h>
#include <stdio.h>

/* clang -Wall simple_query.c -lm */

const double EULER_NUMBER = 2.71828;
const int INODES = 3;
const int HNODES = 4;
const int ONODES = 3;
double Wih[HNODES][INODES] = {
    {0.9, 0.3, 0.4},
    {0.2, 0.8, 0.2},
    {0.1, 0.5, 0.6},
    {0.9, 0.7, 0.9},
};

double Who[ONODES][HNODES] = {
    {0.3, 0.7, 0.5, 0.6},
    {0.6, 0.5, 0.2, 0.3},
    {0.8, 0.1, 0.9, 0.4},
};

void mvp(int rows, int cols, double *matrix, double *vector, double *resvec) {
  /* Matrix-vector product */
  for (int i = 0; i < cols; i++) {
    for (int j = 0; j < rows; j++) {
      resvec[j] += *((matrix + j * cols) + i) * vector[i];
    }
  }
}

void printv(double *vec, int length) {
  for (int i = 0; i < length; i++) {
    printf("%.3f\t", vec[i]);
  }
  putchar('\n');
}

double sigmoid(double n) { return (1 / (1 + pow(EULER_NUMBER, -n))); }
void map(double (*func)(double), int length, double *vector, double *result) {
  for (int i = 0; i < length; i++) {
    result[i] = (*func)(vector[i]);
  }
}
```

```
/* query the neural network */
void query(double *inputs) {
  /* calculate signals into hidden layer */
  double hidden_inputs[HNODES] = {0};
  mvp(HNODES, INODES, (double *)Wih, inputs, hidden_inputs);
  /* calculate the signals emerging from hidden layer */
  double hidden_outputs[HNODES] = {0};
  map(sigmoid, HNODES, hidden_inputs, hidden_outputs);
  /* calculate signals into final output layer */
  double final_inputs[ONODES] = {0};
  mvp(ONODES, HNODES, (double *)Who, hidden_outputs, final_inputs);
  /* calculate the signals emerging from final output layer */
  double final_outputs[ONODES] = {0};
  map(sigmoid, ONODES, final_inputs, final_outputs);
  printf("final_outputs:\n");
  printv(final_outputs, ONODES);
}

int main() {
  double inputs[INODES] = {0.9, 0.1, 0.8};
  query(inputs);
  return 0;
}
```

Компіляція файлу з вихідним кодом і запуск на виконання:

```
$ clang -Wall simple_query.c -lm
$ ./a.out
final_outputs:
0.814   0.757   0.830
```

Для того, щоб сформувати відгук шару на вхідний сигнал, необхідно застосувати до суми (Σ) вхідних даних, помножених на вагові коефіцієнти, функцію активації σ, яка перетворює Σ у вихідний сигнал нейрона. Саме це завдання виконує функція `map()`, якій в якості першого параметра передається вказівник на функцію `sigmoid()`.

Функція `query()` приймає вхідні дані нейронної мережі через параметр, виконує необхідні розрахунки і друкує результат.

Давайте тепер підсумуємо все те, що на цей час вже зроблено: розраховано проходження сигналу через вихідний та прихований шари, тобто визначено значення сигналів на виходах цих шарів. Для повної ясності уточнимо, що ці

значення були отримані шляхом застосування функції активації до комбінованих вхідних сигналів відповідного шару.

Варто запам'ятати таке: незалежно від кількості шарів у нейронній мережі, обчислювальна процедура для кожного з них однакова – комбінування вхідних сигналів, згладжування сигналів для кожного зв'язку між вузлами за допомогою вагових коефіцієнтів та отримання вихідного сигналу за допомогою функції активації. Для нас несуттєво те, скільки шарів утворюють нейронну мережу – 3 або, наприклад, 203, адже до кожного з них застосовується один і той самий підхід.

Повністю зв’язані нейронні мережі (fully connected feedforward artificial neural network), також відомі як багатошарові перцептрони (multilayer perceptron MLP), складаються що щонайменше з трьох шарів нейронів (вхідний, вихідний і принаймні один прихований шар), які перетворюють вхідні дані у вихідні через послідовність зважених сум та нелінійних функцій активації. У повністю зв’язаних шарах кожен нейрон має зв'язок з усіма нейронами попереднього шару, що дозволяє ефективно навчати моделі для різних завдань, включаючи класифікацію зображень.

Глава 3. Нейронна мережа для класифікації зображень рукописних цифр

3.1 База даних рукописних цифр MNIST

На цей момент ми вже вміємо отримувати відгук нейронної мережі на заданий вхідний сигнал і тепер можемо перейти до вирішення наступного завдання – навчимо нейронну мережу розпізнавати рукописні цифри.

Розпізнавання цифр, написаних від руки, відноситься до галузі використання можливостей штучного інтелекту, оскільки ця проблема справді нетривіальна. Вона не така ясна і чітко визначена, як, наприклад, обрахунок добутку матриць, який було зроблено раніше. Класифікацію вмісту зображень за допомогою комп'ютера називають розпізнаванням образів і вирішальну роль тут зіграли технології нейронних мереж. Про існування складнощів можна судити хоча б по тому, що навіть ми, люди, іноді не можемо зрозуміти, яке саме зображення ми бачимо. Зокрема, проблему може викликати написана нерозбірливим почерком буква чи цифра. Вам немає необхідності бути експертом у галузі обчислень або лінійної алгебри, щоб створити програму, яка розпізнає рукописні цифри.

Ми з вами далеко не перші, хто зацікавився цією проблемою. Існує колекція зображень рукописних цифр, які використовуються дослідниками штучного інтелекту як популярний набір для тестування ідей та алгоритмів. Те, що колекція всім відома і користується популярністю, означає, що будь хто може перевірити те, як виглядає його чергова ідея в порівнянні з ідеями інших людей. Тобто різні ідеї та алгоритми тестуються з використанням одного і того ж тестового набору. Ось і ми теж скористаємося цим тестовим набором, щоб перевірити результати наших зусиль щодо створення класифікатора зображень рукописних цифр.

Таким тестовим набором є база даних рукописних цифр під назвою "MNIST". Ця база надається авторитетним дослідником у галузі нейронних мереж на ім'я Yann LeCun для безкоштовного загального доступу за адресою https://yann.lecun.com/exdb/mnist/. За бажанням ви знайдете на цьому сайті відомості щодо успішності колишніх і нинішніх спроб коректного розпізнавання рукописних символів і вже наприкінці нашої з вами роботи, ви зможете порівняти отримане нами рішення з попередніми спробами.

Для завдання класифікації зображень рукописних цифр із набору даних MNIST використовується повністю зв'язана нейронна мережа. Набір даних (dataset) MNIST складається з 60 тисяч навчальних і 10 тисяч тестових зображень розміром 28x28 пікселів, кожне з яких представляє одну з десяти можливих цифр від 0 до 9. Архітектура мережі включає вхідний шар, який представляє кожне зображення у вигляді одновимірного масиву (вектор) довжиною 784 елементи, один прихований шар та вихідний шар з 10 нейронами.

Розбиратися з форматом бази даних MNIST ми з вами зараз не будемо, а скористаємося заздалегідь підготовленими CSV-файлами, в яких окремі значення є звичайним текстом і розділені комами. Ці значення можна переглядати в будь-якому текстовому редакторі, і більшість електронних таблиць або програм, призначених для аналізу даних, можуть працювати з CSV-файлами. Це досить універсальний формат.

Тренувальний набір (https://www.pjreddie.com/media/files/mnist_train.csv) містить 60 тисяч промаркованих екземплярів, що використовуються для тренування нейронної мережі. Слово “промарковані” тут означає, що для кожного екземпляра вказана відповідна правильна відповідь.

Тестовий набір (https://www.pjreddie.com/media/files/mnist_test.csv) включає 10 тисяч екземплярів і використовується для перевірки правильності роботи тренованої нейронної мережі. Кожен рядок набору також містить коректні маркери, що дозволяють побачити, чи здатна нейронна мережа дати правильну відповідь.

Використання незалежних один від одного наборів тренувальних та тестових даних гарантує, що з тестовими даними нейронна мережа раніше не стикалася.

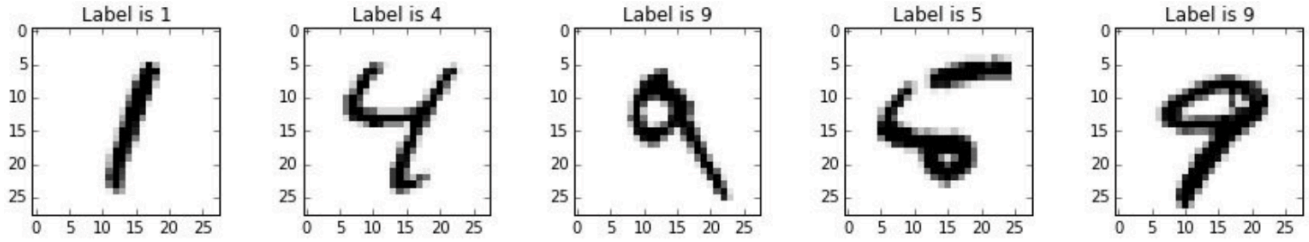
У цих файлах зберігаються довгі рядки тексту, які містять числа, розділені комами. В цьому можна переконатися, відкривши такий файл у текстовому редакторі. Текстові рядки досить довгі, тому кожен із них займає кілька рядків на екрані.

Зміст цих рядків тексту зрозуміти нескладно. Перше значення у рядку – це маркер , тобто фактична цифра, наприклад “7” або “5”, або якась інша з діапазону від “0” до “9”, яку представляє даний рукописний зразок. Цей маркер є правильною відповіддю (label), отриманню якої має навчитися нейронна мережа. Наступні значення, розділені комами, – це пікселі графічного зображення рукописної цифри. Піксельний масив має розмірність 28x28, тому за кожним

маркером слідують 784 пікселі, точніше буде сказати, що ці 784 числа – це коди кольорів пікселів, з яких складається зображення.

Якщо цікаво побачити, як довгий список із 784 значень формує зображення, наприклад, рукописної цифри “5”, то ви повинні сформувати у графічному вигляді ці цифри та переконатися в тому, що вони дійсно є пікселями рукописної цифри. Придивіться уважно і ви помітите, що значення не виходять за межі діапазону від 0 до 255. Ви можете перевірити інші записи, щоб переконатися в тому, що ця умова виконується і для них. Так воно і є: значення кодів усіх кольорів потрапляють у діапазон чисел від 0 до 255. Ймовірно, це цікаво, але відволікатися на це зараз не будемо.

Ось приклади зображень таких рукописних цифр, які присутні в наборах даних:



Глибоке навчання (Deep learning) – це різновид машинного навчання, в рамках якого штучні нейронні мережі навчаються на великих обсягах даних. Воно включає використання багаторівневої апроксимації нелінійних функцій, зазвичай у вигляді нейронних мереж, тобто це набір технік та методів використання нейронних мереж для виконання завдань машинного навчання.

Хочу нагадати, що ця книга не є підручником з глибокого навчання, а використовує приклади з нейронними мережами для демонстрації процесу розробки програм мовою С. Якщо ви вже намагалися дізнатися щось про нейронні мережі та глибоке навчання, то, швидше за все, зіткнулися з достатком ресурсів, від блогів до масових відкритих онлайн-курсів різної якості і навіть книг. Книг з глибокого навчання на даний момент написано вже дуже багато і тільки лінивий ще нічого не написав про те, як створюються і навчаються нейронні мережі. Саме тому переказувати вкотре те, що вже було неодноразово написано багатьма авторами, не має менсу.

Деякі ресурси стосуються в основному концептуальної та математичної частини та містять малюнки, які, як правило, зустрічаються у поясненнях нейронних мереж, а також докладні математичні пояснення того, що

відбувається, щоб ви могли зрозуміти суть. Прикладом цього є хороша книга Ian Goodfellow та ін. «Deep Learning» https://www.deeplearningbook.org/ І тим не менш, давати поради стосовно того, яку саме книгу слід прочитати, немає сенсу. У виданих книгах одні й ті самі принципи пояснюються авторами по-різному з тим чи іншим ступенем уваги до деталей. Для когось одне пояснення може бути зрозумілішим, ніж інше, тому пошукайте і спробуйте читати книги різних авторів. Ви самі відчуєте, яка з цих книг вам краще “заходить”.

3.2. Підготовка даних

Для створення повністю зв'язаної нейронної мережі для класифікації зображень рукописних цифр з набору даних MNIST можна використовувати найрізноманітніші засоби, але для навчальних цілей поки що будемо задовольнятися виключно можливостями, які надає стандартна бібліотека мови C.

Отже, є намір використати дані з файлів MNIST для навчання нейронної мережі, але перш ніж надавати дані мережі, необхідно їх трохи підготувати. Якщо на даний момент ви все ще смутно уявляєте собі те, як працює нейронна мережа, тоді просто прийміть мої слова на віру: нейронні мережі працюють краще, якщо вхідні та вихідні дані конфігуруються таким чином, щоб вони залишалися в діапазоні значень, оптимальному для функцій активації вузлів нейронної мережі. Тому перше, що треба зробити, – це перевести значення кодів кольорів з діапазону значень від 0 до 255 у набагато менший, що охоплює значення від 0,01 до 1,0. Свідомо вибираємо значення 0,01 як нижню межу діапазону, щоб уникнути проблем із нульовими вхідними значеннями, оскільки вони можуть штучно блокувати оновлення ваг. Не існує необхідності вибирати значення 0,99 в якості верхньої межі допустимого діапазону, оскільки немає потреби уникати значень 1,0 для вхідних сигналів. Розподіл вихідних значень, що змінюються в діапазоні від 0 до 255, на 255 приведе їх до діапазону від 0 до 1,0. Подальше множення цих значень на коефіцієнт 0,99 приведе їх до діапазону від 0,0 до 0,99. А тепер інкрементуємо їх значення на 0,01, щоб помістити їх у бажаний діапазон від 0,01 до 1,0.

Маємо тренувальний набір чисел, який містить 60 тисяч промаркованих екземплярів, що буде використовуватися для тренування нейронної мережі, і тепер потрібно перетворити список чисел, розділених комами, у відповідний масив, розбивши довгий текстовий рядок значень, на окремі значення,

використовуючи символ коми як роздільник. Всі ці дії реалізує наступний код мовою C.

```
int readcsv(const char *csvfile, int rows, int cols, double *arr) {
  const size_t MAX_LEN = 2 + 28 * 28 * 4;
  FILE *fd = fopen(csvfile, "r");
  if (fd == NULL) {
    fprintf(stderr, "Error reading file.\n");
    return 1;
  }
  double(*M)[rows][cols] = (void *)arr;
  char buf[MAX_LEN] = {0};
  for (int i = 0; i < rows; i++) {
    char *result = fgets(buf, MAX_LEN, fd);
    if (result != NULL) {
      int j = 0;
      char *c = strtok(result, ",");
      while (c != NULL) {
        (*M)[i][j] = (j == 0) ? atoi(c) : atoi(c) / 255.0 * 0.99 + 0.01;
        j++;
        c = strtok(NULL, ",");
      }
    } else {
      break;
    }
  }
  fclose(fd);
  return 0;
}
```

Для того, щоб скористатися цією функцією, необхідно зарезервувати пам’ять для збереження тренувальних даних. Також було б корисно з’ясувати розмір набору даних (датасету) і вивести його на екран у зручній для сприйняття формі. Це можна зробити за допомогою функції `human_size`:

```
#define TRAIN_FILE "mnist_dataset/mnist_train.csv"
#define TRAIN_ROWS 60000
#define DATA_COLS (1 + 28 * 28)

static const char *human_size(uint64_t bytes) {
  char *suffix[] = {"B", "KB", "MB", "GB", "TB"};
  char length = sizeof(suffix) / sizeof(suffix[0]);
  int i = 0;
```

```
  double dbytes = bytes;
  if (bytes > 1024) {
    for (i = 0; (bytes / 1024) > 0 && i < length - 1; i++, bytes /= 1024) {
      dbytes = bytes / 1024.0;
    }
  }
  static char output[200];
  sprintf(output, "%.02lf %s", dbytes, suffix[i]);
  return output;
}
. . .
uint64_t bytes = (TRAIN_ROWS * DATA_COLS) * sizeof(double);
double *ptrtd = (double *)malloc(bytes);
/* Check if the memory has been successfully allocated by malloc or not */
if (ptrtd == NULL) {
  fprintf(stderr, "Memory not allocated.");
  exit(EXIT_FAILURE);
}
printf("Training data: %s\n", TRAIN_FILE);
readcsv(TRAIN_FILE, TRAIN_ROWS, DATA_COLS, (double *)ptrtd);
/* PRIu64 is a format specifier, introduced in C99, for printing uint64_t */
printf("Size of training data %" PRIu64 " Bytes: %s\n", bytes,
human_size(bytes));
```

При виконанні цього фрагменту коду отримаємо наступне:

```
Training data: mnist_dataset/mnist_train.csv
Size of training data 376800000 Bytes: 359.34 MB
```

Звертаю вашу увагу на те, що нейронні мережі навчаються на великих обсягах даних, іноді дуже великих. Дані, які використовуються для виконання завдання класифікації рукописних цифр, відносно невеликі, вони цілком можуть розміститися в оперативній пам'яті і не вимагають спеціальних заходів щодо їх обробки. Робота з великими обсягами тренувальних даних виходить за рамки теми цієї книги і тут не розглядається.

Тепер саме час задати кількість вузлів вхідного, прихованого та вихідного шарів. Кількість вузлів вхідного шару 784. Цю цифру ми детально обговорили вище. Кількість вузлів вихідного шару вибрано 10 з міркувань того, що мережа розпізнаватиме рукописні цифри з діапазону від 0 до 9. Інакше кажучи, очікується, що нейронна мережа класифікує зображення і надасть йому коректний маркер. Таким маркером може бути одне з десяти чисел в діапазоні від

0 до 9. Це означає, що вихідний шар мережі повинен мати 10 вузлів, по одному на кожну можливу відповідь, або маркер. Якщо відповіддю є “0”, то активізуватися повинен перший вузол, тоді як інші вузли мають бути пасивними. Якщо відповіддю є “9”, то активізуватися повинен останній вузол вихідного шару при інших пасивних вузлах. Кількість вузлів прихованого шару вибрано 200. Будемо поки вважати, що цей вибір був нами зроблений довільно. Ці дані визначають конфігурацію та розмір нейронної мережі:

```
/* number of input, hidden and output nodes */
const int INODES = 784;
const int HNODES = 200;
const int ONODES = 10;
```

Немає на цей момент ніякого суворого наукового обґрунтування вибору двох сотень прихованих вузлів. Це число було вибрано меньшим, ніж 784, з тих міркувань, що нейронна мережа має знаходити у вхідних значеннях такі особливості або шаблони, які можна виразити у більш короткій формі, ніж ці значення. Тому, вибираючи кількість вузлів меншою, ніж кількість вхідних значень, ми змушуємо мережу намагатися знаходити ключові особливості шляхом узагальнення інформації. У той же час, якщо вибрати кількість прихованих вузлів занадто малою, буде обмежено можливості мережі щодо визначення достатньої кількості відмітних ознак або шаблонів у зображенні. Тим самим мережа була б позбавлена можливості виносити власні судження щодо даних MNIST. З урахуванням того, що вихідний шар повинен забезпечувати виведення 10 маркерів, а отже, повинен мати десять вузлів, вибір проміжного значення 200 для кількості вузлів прихованого шару є цілком розумним.

Немає ідеального загального методу для вибору кількості прихованих вузлів. Також немає і ідеального методу вибору кількості прихованих шарів. На цей час найкращим підходом є проведення експериментів доти, доки не буде отримана конфігурація мережі, оптимальна для завдання, яке ви намагаєтесь вирішити. Саме цими експериментами вам пропонується зайнятися після того, як ми закінчимо вирішувати це завдання.

3.3. Ініціалізація мережі

Найважливіша частина нейронної мережі – матриці вагових коефіцієнтів зв'язків (ваги). Ці матриці використовуються для розрахунку поширення сигналів

у прямому напрямку, а також зворотного розповсюдження помилок, і саме вагові коефіцієнти уточнюються у спробі покращити характеристики мережі. Подивіться ще раз уважно на графік функції активації: деякі надто великі ваги можуть змістити функцію активації в область великих значень, що призвело б до її насичення. Тому слід уникати великих початкових значень вагових коефіцієнтів, оскільки використання функції активації в цій області значень може призводити до насичення мережі та зниження здатності мережі вчитися на кращих значеннях. Найгірший вибір початкових значень вагових коефіцієнтів – нульові значення, оскільки вони повністю знищують вхідний сигнал. У цьому випадку функція оновлення ваг, яка залежить від вхідних сигналів, обнуляється, тим самим повністю виключаючи можливість оновлення ваг. Значення вагових коефіцієнтів внутрішніх зв'язків мають бути випадковими та невеликими і можуть мати як позитивні, так і негативні значення і змінюватися в діапазоні від -1,0 до +1,0. Для простоти віднімемо 0,5 із цих граничних значень, перейшовши до діапазону значень від -0,5 до +0,5.

Створення матриць початкових ваг здійснюється за допомогою функції `fill_random`:

```
double Wih[HNODES][INODES] = {0};
double Who[ONODES][HNODES] = {0};
void fill_random(double *arr, int rows, int cols) {
  srand(time(NULL));
  for (int i = 0; i < rows; i++) {
    int row = i * cols;
    for (int j = 0; j < cols; j++) {
      *((arr + row) + j) = 1.0 * rand() / RAND_MAX - 0.5; /* -0.5 .. +0.5 */
    }
  }
}
. . .
fill_random((double *)Wih, HNODES, INODES);
fill_random((double *)Who, ONODES, HNODES);
```

В літературі на тему глибокого навчання пишуть також про інші різні способи підготовки даних та ініціалізації вагових коефіцієнтів. Існує дещо вдосконалений підхід до створення випадкових початкових значень ваг. Для цього вагові коефіцієнти вибираються з нормального розподілу з центром у нулі та зі стандартним відхиленням, величина якого обернено пропорційна кореню

квадратному з кількості вхідних зв'язків на вузол. Пропоную вам зробити це самостійно. Google вам допоможе.

Було б корисно поцікавитися розмірами пам'яті, яку займають матриці вагових коефіцієнтів:

```
uint64_t bytes = (HNODES * INODES) * sizeof(double); // sizeof(Wih);
printf("Size of Wih %" PRIu64 " Bytes: %s\n", bytes, human_size(bytes));
bytes = (ONODES * HNODES) * sizeof(double); // sizeof(Who);
printf("Size of Who %" PRIu64 " Bytes: %s\n", bytes, human_size(bytes));
```

При виконанні цього фрагменту коду отримаємо наступне:

```
Size of Wih 1254400 Bytes: 1.20 MB
Size of Who 16000 Bytes: 15.62 KB
```

Зрозуміло, що розмір пам'яті, яку займають ці дві матриці, відносно невеликий і не потребує додаткової уваги.

3.4. Тренування нейронної мережі

На цей момент ми вже вміємо обчислювати вихідні сигнали нейронної мережі за заданими величинами вхідних сигналів. Наступний крок полягає у порівнянні вихідних сигналів нейронної мережі з даними тренувального прикладу для визначення помилки. Помилка нейронної мережі є функцією ваг внутрішніх зв'язків. Потрібно знати величину цієї помилки, щоб можна було покращити вихідні результати шляхом налаштування параметрів мережі. Інакше кажучи, потрібно десь, щось трохи підкрутити у самій мережі, щоб вона правильно розпізнавала рукописні цифри. Підкрутити можна відповідні ваги, але які саме і наскільки – це головне питання. Безпосередній підбір відповідних ваг, наприклад, методом грубої сили наштовхується на значні труднощі. Альтернативний підхід полягає у ітеративному поліпшенні вагових коефіцієнтів шляхом зменшення функції помилки невеликими кроками. Кожен крок здійснюється у напрямку якнайшвидшого спуску з поточної позиції. Цей підхід називається градієнтним спуском. Градієнт помилки розраховується за допомогою диференційного обчислення.

Традиційно будь-яке програмне забезпечення, що реалізує такі когнітивні здібності, як сприйняття, пошук, планування та навчання, є частиною штучного інтелекту. Нейронну мережу можна сприймати як універсальний апроксиматор функцій, який теоретично може представити вирішення майже будь-якої

контрольованої проблеми навчання. Однією з фундаментальних концепцій навчання мережі є зворотне поширення (backpropagation) помилки – метод градієнтної оцінки, який використовується для навчання моделей нейронної мережі. Оцінка градієнта використовується алгоритмом оптимізації для обчислення оновлень параметрів мережі. Якщо вам ці слова видалися страшними і незрозумілими, то зараз саме час поцікавитися тим, як працює алгоритм зворотного поширення помилки. Нічого такого, що потребувало б нелюдських розумових зусиль для розуміння ідеї цього алгоритму, насправді немає. Коротше кажучи, нейронні мережі навчаються уточненням вагових коефіцієнтів своїх зв'язків. Цей процес керується помилкою – різницею між правильною відповіддю, що надається тренувальними даними, і фактичним вихідним значенням нейронної мережі. Помилка на вихідних вузлах визначається простою різницею між бажаним і фактичним вихідними значеннями. У той самий час величина помилки, яка повязана з внутрішніми вузлами, менш очевидна. Одним із способів вирішення цієї проблеми є розподіл помилок вихідного шару між відповідними зв'язками пропорційно до ваги кожного зв'язку з наступним об'єднанням відповідних розрізнених частин помилки на кожному внутрішньому вузлі. Це є сутність алгоритму зворотного поширення. Все це описано і детально розтлумачено майже у кожній книзі, присвяченій темі глибокого навчання, тому опису алгоритму зворотного поширення у цій книзі немає. Розбираючись з цим алгоритмом, зверніть увагу на те, що зворотне поширення помилок описується за допомогою вже знайомого вам матричного множення. Вміння розраховувати зворотне розповсюдження помилок до кожного шару мережі дозволяє зрозуміти те, як мають бути змінені ваги зв'язків, щоб покращити загальну результуючу відповідь на виході нейронної мережі. Вихідний сигнал нейронної мережі є складною функцією з багатьма параметрами, ваговими коефіцієнтами зв'язків, які впливають на вихідний сигнал. Метою навчання нейронної мережі на тренувальних даних є отримання цієї функції.

Вихідні сигнали повинні знаходитися в межах діапазону, який може забезпечити функція активації. Значення, менші або рівні 0 або більші або рівні 1, не сумісні з логістичною сигмоїдою. Установка тренувальних цільових значень за межами допустимого діапазону призведе до ще більших значень ваг і, зрештою, до насичення мережі. Підходящим варіантом є діапазон значень від 0,01 до 0,99. Якщо тренувальний приклад позначений маркером “5”, то для вихідного вузла слід створити такий цільовий масив, в якому будуть малі значення всіх елементів,

крім одного, що відповідає маркеру “5”. У разі цей масив міг би виглядати приблизно так: `[0,0,0,0,0,1,0,0,0,0]`.

Насправді ці числа потребують додаткового масштабування, оскільки ми вже бачили, що спроби створення на виході нейронної мережі значень 0 і 1, недосяжні через використання функції активації, і призводять до великих ваг і насичення мережі. Отже, натомість використовуватимемо значення 0,01 і 0,99, і тому цільовим масивом для маркера “5” має бути масив

```
[0.01, 0.01, 0.01, 0.01, 0.01, 0.99, 0.01, 0.01, 0.01, 0.01]
```

Всі ці міркування реалізує наступний код мовою C.

```
double(*TD)[TRAIN_ROWS][DATA_COLS] = (void *)ptrtd;

for (int i = 0; i < TRAIN_ROWS; i++) {
  double *ptrd = (*TD)[i];
  double targets[ONODES] = {[0 ... ONODES - 1] = 0.01};
  targets[(int)(*ptrd)] = 0.99;
  ptrd++; // skip 0-element; now ptrd is a pointer to inputs
  train(ptrd, targets);
}
```

Цей рядок коду вибирає перший елемент запису з набору даних MNIST, що є цільовим маркером тренувального набору:

```
double *ptrd = (*TD)[i];
```

Згадайте, що запис читається з вихідного файлу у вигляді текстового рядка, а не числа. Як тільки перетворення виконано, отриманий цільовий маркер використовується для встановлення значення відповідного елемента масиву рівним 0,99. Наприклад, маркер “0” буде перетворено на ціле число 0, що є коректним індексом даного маркера в масиві:

```
targets[(int)(*ptrd)] = 0.99;
```

У тілі циклу присутній виклик функції `train`, якій передаються два параметри: масив тренувальних даних `ptrd` і масив `targets`, точніше буде сказати, що передаються вказівники, хоча це суті справи не змінює. Функція `train` реалізує процес тренування нейронної мережі, використовуючи

тренувальні дані, що надаються їй на кожному кроці циклу. Усі необхідні для реалізації процесу тренування функції розташовані в окремому у файлі `common.c`.

3.5. Складання проекту на мові С

У програмуванні все починається з вихідного коду. Насправді вихідний код, який ще іноді називають кодовою базою, зазвичай складається з цілого ряду текстових файлів. Кожен файл містить інструкції, написані мовою програмування. У цьому підрозділі демонструється, як зібрати проект, написаний на С. Приклад, з яким ми будемо працювати, складається з кількох вихідних файлів, що характерно майже для всіх проектів цією мовою. Але перш ніж переходити до збірки, спочатку розглянемо структуру нашого проекту на С.

Будь-який проект на С містить вихідний код (або кодову базу) та інші документи, які описують розроблюваний додаток і використовувані стандарти. Код С зазвичай зберігається у файлах двох типів: у заголовкових файлах, які мають розширення .h; у вихідних файлах із розширенням .c. Для стислості заголовкові файли будуть в подальшому називатися заголовками.

Заголовок зазвичай містить макроси та визначення типів, а також оголошення функцій, глобальних змінних та структур. У мові програмування С оголошення та визначення деяких елементів програмування, таких як функції, змінні та структури, можуть знаходитись у різних файлах. Оголошення функції показує, як її використовувати, а визначення містить її реалізацію. Оголошення функцій рекомендується розміщувати в заголовках, а їх визначення – у відповідних вихідних файлах. Це особливо стосується функцій. І хоча це не є обов'язковою вимогою, такий підхід до проектування дозволяє зберігати визначення функцій поза заголовками. До вихідних файлів можна підключати лише заголовки. Заголовки не повинні містити нічого, крім оголошень. До вихідного файлу не підключаються інші вихідні коди, тому він завжди компілюється окремо. Пам'ятайте: відповідно до загальноприйнятих рекомендацій, вихідні файли C/C++ підключати не можна. (Мається на увазі за допомогою директиви `#include`).

Подивимося тепер на те, як усе влаштовано у нашому проекті нейронної мережі розпізнавання рукописних цифр. Створений на цей час код, розміщено в трьох файлах: одному заголовку і двох файлах з вихідним кодом. Усі вони знаходяться в одному каталозі:

```
$ ls -l
. . .
-rw-r--r--  1 vadim  staff     8057 Jul 29 20:21 common.c
-rw-r--r--  1 vadim  staff     1589 Jul 29 20:20 common.h
-rw-r--r--  1 vadim  staff     4118 Jul 29 20:21 train.c
```

Я підкреслю ще раз: оголошення функцій зберігаються у заголовковому файлі, а визначення (або тіла) у вихідному. Порушувати це правило можна лише в окремих випадках. Крім того, щоб мати доступ до оголошення, вихідний файл повинен підключити заголовковий файл. Саме так це працює у C та C++.

Отже, наш проект складається з кількох файлів, тому зараз необхідно розібратися у тому, як правильно скомпілювати такий проект. Заголовковий файл відіграє роль містка, що сполучає два вихідні файли і дозволяє розділити код на дві частини, які збираються разом. Наведений нижче заголовковий файл `common.h` містить усе, що потрібно одному вихідному коду для використання функціональності іншого.

```
/* File common.h */
#ifndef COMMON_H
#define COMMON_H
#include <stdint.h> /* uint64_t */
#include <stdio.h>  /* size_t */
#define LOG_ERROR(format, ...) fprintf(stderr, format, __VA_ARGS__)
/* number of input, hidden and output nodes */
#define INODES 784
#define HNODES 200
#define ONODES 10
double Wih[HNODES][INODES];
double Who[ONODES][HNODES];
const char *human_size(uint64_t bytes);
int readcsv(const char *csvfile, int rows, int cols, double *arr);
. . .
void train(double *inputs, double *targets);
. . .
#endif
```

Тут можна побачити попередні оголошення функцій. Попереднім називається оголошення, що знаходиться перед відповідним визначенням. Крім того, тут також застосовується запобігання дублюванню, яке не дає компілятору підключити заголовковий файл двічі. У будь-якому проекті мовою C функція `main` служить точкою входу у програму. Ця функція знаходиться у файлі `train.c`.

Файли, з якими ви познайомилися вище, потрібно зібрати, щоб отримати результуючий файл, тобто файл, який можна буде запустити на виконання. Складання проекту на C/C++ вимагає компіляції його кодової бази в об'єктні файли, які ще називають проміжними, і потім об'єднання їх в кінцеві продукти, такі як статичні бібліотеки або виконувані файли. Для компіляції представлених тут вихідних файлів ми не будемо використовувати інтегроване середовище розробки (Integrated Development Environment, IDE). Натомість ми застосуємо компілятор безпосередньо, без допоміжного програмного забезпечення. Описані кроки нічим не відрізняються від тих, які фоново виконує IDE, компілюючи набір вихідних файлів.

Компіляція набору вихідних файлів:

```
$ clang -Wall train.c common.c -lm -o nn
```

3.6. Додаткові пояснення

Тіло функції `main` файлу `train.c` містить код, який потребує додаткових пояснень.

```
int main(void) {
  clock_t begin = clock();
  . . .
  /* train the neural network */
  /* epochs is the number of times the training data set is used for training */
  int epochs = 5;
  double(*TD)[TRAIN_ROWS][DATA_COLS] = (void *)ptrtd;
  for (int e = 0; e < epochs; e++) {
    printf("epoch %d\n", e);
    for (int i = 0; i < TRAIN_ROWS; i++) {
      . . .
      ptrd++; // skip 0-element; now ptrd is a pointer to inputs
      train(ptrd, targets);
    }
  }
  free(ptrtd);
  const char *wihname = getname("wih", HNODES, INODES);
  tocsv(wihname, HNODES, INODES, Wih);
  const char *whoname = getname("who", ONODES, HNODES);
  tocsv(whoname, ONODES, HNODES, Who);
  double time_spent = (double)(clock() - begin) / CLOCKS_PER_SEC;
```

```
  printf("Time spent:\t%.2f seconds.\n", time_spent);
  return 0;
}
```

Навчальний набір даних використовується для навчання кілька разів, тобто здійснюється багаторазове повторення циклів тренування з тим самим набором даних. Щодо одного тренувального циклу іноді використовують термін “епоха”. Тому сеанс тренування з п'яти епох означає п'ятиразовий прогін всього тренувального набору даних. А навіщо це робити особливо якщо для цього комп'ютеру потрібно більше часу? Причина полягає в тому, що тим самим ми намагаємося забезпечити більше маршрутів градієнтного спуску, що оптимізують вагові коефіцієнти. Це звичайна практика. Подробиці і обґрунтування ви за бажанням знайдете в інтернеті. Подивимося, що нам дадуть п'ять тренувальних епох. Ми виконали компіляцію набору вихідних файлів і отримали в результаті виконуваний файл, який можемо запустити і оцінити кількість часу, витраченого на виконання тренування:

```
$ ./nn
Training data: mnist_dataset/mnist_train.csv
Size of training data 376800000 Bytes: 359.34 MB
epoch 0
epoch 1
epoch 2
epoch 3
epoch 4
Time spent:     385.85 seconds.
```

Ви не забули про те, що матриці вагових коефіцієнтів, елементи яких було змінено в процесі тренування, все ще знаходяться в оперативній пам’яті? Після закінчення процесу тренування, слід потурбуватися про те, щоб зберегти результати тренування на диску. Це надасть можливість в подальшому користуватися натренованою мережею. Збереження матриці у вигляді CSV-файлу на диску здійснює функція `tocsv`. Очевидно, що має бути функція, яка вміє відновлювати вміст матриці в оперативній пам'яті з файлу на диску. Така функція є. Це функція `fromcsv`:

```
void tocsv(const char *csvfile, size_t rows, size_t cols,
       double arr[rows][cols]);
void fromcsv(const char *csvfile, size_t rows, size_t cols,
       double arr[rows][cols]);
```

У мові C є тип `size_t`, який зазвичай використовується для представлення розміру об’єктів у байтах і тому використовується як тип повернення оператором `sizeof`. Він гарантовано буде достатньо великим, щоб вмістити розмір найбільшого об’єкта, який може бути оброблений вашим комп’ютером. В основному максимально допустимий розмір залежить від компілятора; якщо компілятор 32-розрядний, то це просто `typedef` (тобто псевдонім) для `unsigned int`, але якщо компілятор 64-бітний, то це буде `typedef` для `unsigned long long`. Тип даних `size_t` ніколи не є негативним. Тому багато функцій бібліотеки C такі, наприклад, як `malloc`, `memcpy` і `strlen`, оголошують свої аргументи з типом повернення як `size_t`. Це натяк на те, що в коді використовується в деяких місцях тип `int`, хоча правильніше було б використовувати тип `size_t`. В файлі `common.c` ви знайдете згадані дві функції, в яких було в якості приклада використано тип `size_t`. Для вас було б дуже корисно проаналізувати вже написаний код і внести редагування, змінивши тип `int` на `size_t` там, де це необхідно.

У тілі функції `train()` використовується коефіцієнт навчання - це множник, що згладжує величину змін, щоб уникнути перельотів за мінімум у методі градієнтного спуску. Коефіцієнт навчання можна налаштовувати з урахуванням особливостей конкретного завдання.

```
const float LEARNING_RATE = 0.1;
```

До речі, ви пам'ятаєте, що код написано на C із застосуванням засобів лише стандартної бібліотеки? З огляду на це, зовсім не дивує те, що у разі сучасних швидкодіючих домашніх комп'ютерів, а використано було MacBook Pro, обробка всіх 60 тисяч тренувальних прикладів, для кожного з яких необхідно обчислити поширення сигналів від 784 вхідних вузлів через двісті прихованих вузлів у прямому напрямку, а також зворотне поширення помилок і оновлення ваг, зайняло всього кілька хвилин, точніше 386 секунд. Для вашого комп'ютера тривалість обчислень може бути іншою.

Спробуйте повторити це на іншій мові програмування. А поки ви це будете робити, ми продовжимо писати на C, тому наш легкий для читання, але не ідеально оптимізований код все одно працюватиме на порядок швидше, ніж порівняний код будь-якою іншою мовою, яка розпухла від великої кількості усякої функціональності.

3.7. Тестування нейронної мережі

В результаті виконання процесу тренування мережі, ми маємо збереженим на диску вміст двох матриць: матриці ваг `Wih`, яка містить вагові коефіцієнти для зв'язків між вхідним та прихованим шарами, і матрицю `Who`, яка містить коефіцієнти для зв'язків між прихованим та вихідним шарами. Вміст цих матриць знаходиться у файлах `wih_200_784.csv` і `wih_200_784.csv` відповідно.

Далі ми додамо до нашого набору вихідних файлів новий файл `query.c`, розроблений для тестування ефективності мережі. На цей час набір файлів містить в собі:

```
$ ls -l
-rw-r--r--  1 vadim  staff     8057 Jul 29 20:21 common.c
-rw-r--r--  1 vadim  staff     1589 Jul 29 20:20 common.h
-rw-r--r--  1 vadim  staff     2490 Jul 29 20:46 query.c
-rw-r--r--  1 vadim  staff     1916 Jul 29 20:28 train.c
-rw-r--r--  1 vadim  staff    37261 Jul 29 20:32 who_10_200.csv
-rw-r--r--  1 vadim  staff  2908419 Jul 29 20:32 wih_200_784.csv
```

Фрагментарний вміст файлу `query.c` представлено нижче:

```
. . .
#define TEST_FILE "mnist_dataset/mnist_test.csv"
#define TEST_ROWS 10000
#define DATA_COLS (1 + 28 * 28)

int idxmax(double *vec, int length) {
  double res = vec[0];
  int idx = 0;
  for (int i = 1; i < length; i++) {
    if (res < vec[i]) {
      res = vec[i];
      idx = i;
    }
  }
  return idx;
}

void viewres(int *vec, int length) {
  for (int i = 0; i < length; i++) {
    printf("%3d\t", vec[i]);
```

```
  }
  putchar('\n');
}

int main() {
  uint64_t bytes = (TEST_ROWS * DATA_COLS) * sizeof(double);
  double *ptrtd = (double *)malloc(bytes);
  /* Check if the memory has been successfully allocated by malloc or not */
  . . .
  printf("Testing data: %s\n", TEST_FILE);
  readcsv(TEST_FILE, TEST_ROWS, DATA_COLS, (double *)ptrtd);
  . . .
  const char *wihname = getname("wih", HNODES, INODES);
  fromcsv(wihname, HNODES, INODES, Wih);
  . . .
  const char *whoname = getname("who", ONODES, HNODES);
  fromcsv(whoname, ONODES, HNODES, Who);
  . . .
  /* scorecard for how well the network performs, initially 0 */
  int success[ONODES] = {0};
  int fail[ONODES] = {0};
  int total = 0;
  double(*TD)[TEST_ROWS][DATA_COLS] = (void *)ptrtd;
  for (int i = 0; i < TEST_ROWS; i++) {
    . . .
    ptrd++; // skip 0-element; now ptrd is a pointer to inputs
    double final_outputs[ONODES] = {0};
    query(ptrd, final_outputs);
    int idx = idxmax(final_outputs, ONODES);
    if (idx == label) {
      success[label]++;
      total++;
    } else {
      fail[label]++;
    }
  }

  /* the performance score, the fraction of correct answers */
  printf("Performance score: %.2f\n", (total / (double)TEST_ROWS));
  viewres(success, ONODES);
  viewres(fail, ONODES);

  free(ptrtd);
  return 0;
```

```
}
```

Функція `getname` формує ім’я файлу, з якого буде відновлено в оперативній памяті попередньо збережену матрицю вагових коефіцієнтів. В свою чергу, функція `fromcsv` відновлює вміст обох матриць в оперативній пам'яті з файлів на диску. Те, як влаштована функція `query` вже було розібрано вище, тому на ній зупинятися не будемо.

Перш ніж увійти в цикл, що обробляє всі записи тестового набору даних, створюється два порожніх масиви `success` і `fail`, які будуть виконувати роль журналу оцінок роботи мережі, і які оновлюються після обробки кожного запису. Потім робиться розрахунок ефективності і функція `viewres` виводить у стандартний потік виводу результати. Ще один суттєвий момент, про який часто забувають, це `free(ptrtd)`. Не будемо забувати звільняти ресурси.

Компіляція файлу з вихідним кодом і запуск на виконання:

```
$ clang -Wall query.c common.c -lm -o query
$ ./query
Testing data: mnist_dataset/mnist_test.csv
Size of testing data 62800000 Bytes: 59.89 MB
Size of Wih 1254400 Bytes: 1.20 MB
Size of Who 16000 Bytes: 15.62 KB
Performance score: 0.97
 973    1124     999     983     954     863     933     987     937     977
   7      11      33      27      28      29      25      41      37      32
```

Останні два рядки виведення містять по десять елементів. Наприклад, число 1124 – це кількість правильно розпізнаних зображень цифри “1” у тестовому наборі, а число 11 – відповідно неправильно розпізнаних цифр “1”.

Відповідно до результатів навчання нашої простої 3-шарової нейронної мережі з використанням набору даних, що включає 60 тисяч прикладів, та подальшого тестування на 10 тисяч записів показник загальної ефективності мережі складає `0.97`. Точність розпізнавання становить 97%. Це досить непогано. Цей показник, що дорівнює 97%, можна порівняти з аналогічними результатами еталонних тестів, які можна знайти за вже відомою адресою http://yann.lecun.com/exdb/mnist/ . Там ви побачите, що в деяких випадках наші результати навіть кращі за еталонні і майже порівняні з наведеними на вказаному сайті результатами для найпростішої нейронної мережі, ефективність якої склала

95,3%. Як для навчального та демонстраційного прикладу, написаного “на коліні” і не відшліфованого до стану продукту, результат дійсно непоганий. Подальше вдосконалення цього коду залишаю вам для вашої самостійної творчості.

Отримання показника ефективності 97% при тестуванні набору даних MNIST нашою нейронною мережею – це зовсім непогано, і ваше бажання зупинитися на цьому можна було б вважати цілком виправданим. Однак давайте спробуємо поекспериментувати. Насамперед, ми можемо спробувати налаштувати коефіцієнт навчання. Перед цим ми задали його рівним `LEARNING_RATE = 0.1`, навіть не тестуючи інші значення. Якщо ми збільшимо це значення до 0,6 і подивимося, як це позначиться на можливості нейронної мережі вчитися, то побачимо, що виконання коду з таким значенням коефіцієнта навчання дає результат гірший від попереднього. Очевидно, збільшення коефіцієнта навчання порушує монотонність процесу мінімізації помилок методом градієнтного спуску та супроводжується перескоками через мінімум. А що станеться, якщо ми зменшимо коефіцієнт навчання до 0,01? Це також призводить до зменшення показника ефективності мережі. Очевидно, занадто малі значення коефіцієнта навчання знижують ефективність. Це є логічним, оскільки малі кроки зменшують швидкість градієнтного спуску. Врахуйте, якщо ви самостійно будете виконувати цей код, то ваші оцінки трохи відрізнятимуться від наведених тут, оскільки процес в цілому містить елементи випадковості. Ваш випадковий вибір початкових значень вагових коефіцієнтів не співпадатиме з моїм, а тому маршрут градієнтного спуску для вашого коду буде іншим.

Подібно до того, як ми налаштовували коефіцієнт навчання, проведемо експеримент з використанням різної кількості епох і оцінимо залежність показника ефективності від цього фактору. Щось підказує, що чим більше тренувань, тим вища ефективність, але можна припустити, що занадто велика кількість тренувань може призвести до погіршення ефективності через так зване перенавчання мережі на тренувальних даних, що знижує ефективність при роботі з незнайомими даними. Слід побоюватися фактора перенавчання у будь-яких видах машинного навчання, а не лише у нейронних мережах. Для студентів перенавчання можливо теж загрожує певними наслідками, тому не слід робити експерименти з перенавчанням власного мозку.

Вихідний код, який було розглянуто в цій главі, а також два CSV-файли з матрицями можна завантажити за адресою на GitHub: https://github.com/iamvadimov/cmnist

Підведемо підсумки. У штучному інтелекті є область, присвячена створенню програмного забезпечення для виконання завдань, які потребують інтелекту, через навчання на основі даних. Вона називається машинне навчання, у якого в свою чергу є три основні напрями: навчання з учителем, без вчителя та навчання з підкріпленням. Навчання з учителем передбачає використання промаркованих даних. У процесі навчання людина вирішує, які дані потрібно зібрати і як їх помітити. Мета цього напрямку машинного навчання – узагальнення. Класичний приклад – створена тут програма для розпізнавання цифр, написаних від руки: одна добра людина зібрала зображення з рукописними цифрами і промаркувала їх, а ми навчили мережу правильно розпізнавати та категоризувати ці цифри. При цьому очікувалось, що навчена мережа зможе узагальнювати та категоризувати зображення з такими цифрами. Судячи з отриманих результатів, створена мережа дійсно може це робити.

Акцентную увагу на наступному. Нейронна мережа це не алгоритм, а структура, яка складається з кількох верств математичних перетворень, що застосовуються до вхідних значень, створення якої було виконано з використанням можливостей лише стандартної бібліотеки мови C, без залучення спеціальних засобів.

Зараз, коли у вас є код, і з'явилося відчуття задоволення від того, що все вийшло, необхідно трохи остудити ваш запал. Насправді, все не так просто, як могло здатися. Зауважте, що для завдання було взято у достатній кількості рафіновані, тобто завчасно дбайливо підготовлені дані, в яких мінімум помилок. Частіше може статися так, що тренувальних даних буде недостатньо для ефективного навчання мережі. У тренувальних даних можуть бути помилки, у зв'язку з чим справедливість припущення про те, що вони істинні і на них можна вчитися, під питанням. Кількість шарів або вузлів у самій мережі може бути недостатньою для того, щоб правильно моделювати рішення задачі тощо. Це означає, що підходи, які ви будете використовувати в подальшому, повинні враховувати зазначені нюанси. Сподіваюся, що у вас вже з'явилися ідеї щодо вдосконалення коду та проведення нових експериментів з більшою кількістю шарів мережі та вузлів.

Глава 4. Бібліотека Glib

4.1. Перш ніж почати

Програми пишуть для маніпулювання даними. Наприклад, ваша програма може читати список імен із файлу, запитувати у користувача деякі дані через графічний інтерфейс користувача (GUI) або завантажувати дані із зовнішнього апаратного пристрою. Як тільки дані з’являться у вашій програмі, ви повинні пильно стежити за ними. Функції та змінні, які ви використовуєте для керування даними, називаються структурами даних або контейнерами. Проблеми з відстеженням даних у C багато разів вирішувались завдяки використанню стандартних контейнерів, таких як зв’язаний список і бінарне дерево. Кожен першокурсник спеціальності інформатика відвідує курс основ програмування і структур даних, на якому викладач обов'язково призначає серію вправ з написання реалізацій цих контейнерів. Під час написання цих структур студент отримує перше розуміння того, наскільки вони складні; dangling pointers, витік памяті (memory leaks) та купа інших "сюрпризів" чекають за кожним рогом, щоб зловити необережного студента. Написання модульних тестів може дуже допомогти, але загалом переписувати одну й ту саму структуру даних для кожної нової програми - нудне заняття і невдячна справа.

Звісно існують вбудовані структури даних, і деякі мови програмування мають вбудовані контейнери. Наприклад, C++ має стандартну бібліотеку шаблонів (STL), яка містить набір класів контейнерів, таких як списки, черги пріоритетів, набори та карти. Ці контейнери також безпечні для типів, тобто ви можете помістити лише один тип елемента в кожен об’єкт-контейнер, який ви створюєте. Це робить їх безпечнішими у використанні та усуває багато стомлюючих перевірок і перетворень даних, яких вимагає C. Також STL містить безліч ітераторів, утиліт сортування тощо, щоб полегшити роботу з контейнерами. Мова програмування Java також постачається з набором класів-контейнерів. Пакет java.util містить ArrayList, HashMap, TreeSet та різні інші стандартні структури. Java містить утиліти для загального сортування даних і створення незмінних колекцій, а також різні інші корисні елементи. Однак у стандартній бібліотеці мови C немає вбудованої підтримки контейнерів і вам доведеться або створити свою власну, або використовувати стронню бібліотеку структур даних. Насправді це зовсім не проблема. На ваше щастя є бібліотека GLib. Це чудова безкоштовна бібліотека з відкритим кодом, яка задовольнить цю

потребу. Вона містить більшість стандартних структур даних і багато утиліт, необхідних для ефективного маніпулювання даними у ваших програмах. Вона існує вже майже пару десятиліть (вона існує з 1996 року), тому її було ретельно протестовано і додано багато корисних функцій. Як бачите, корисні інструменти є. Залишилося зовсім трохи – зрозуміти те, як ними скористатися.

4.2. Короткий огляд GLib

Розглянемо область застосування GLib (https://docs.gtk.org/glib/). Оскільки стандартна бібліотека мови C аж ніяк не всеосяжна, природно, що з часом з'явилися бібліотеки, що заповнюють існуючі прогалини. Відверто кажучи, у бібліотеці GLib реалізовано стільки базових обчислювальних засобів, що вона цілком могла б скласти за вас іспит за перший курс навчання інформатиці. Крім того, вона перенесена практично на всі платформи (включаючи навіть версії MS Windows, що не підтримують POSIX) і досить стабільна, тому на неї можна цілком покластися.

Дещо уточнимо стосовно POSIX (https://uk.wikipedia.org/wiki/POSIX). Все, що так чи інакше пов'язано з файловими системами, можна розбити на два класи, що слабо перекриваються: POSIX-сумісні системи та сімейство операційних систем Windows. Сумісність з POSIX взагалі не означає, що система повинна виглядати та працювати як Unix. Наприклад, типовий користувач Mac поняття не має, що використовує стандартну систему BSD з красивим і розфарбованим фасадом, але знаючі люди можуть перейти в папку Applications - Utilities, відкрити програму Terminal і виконувати милі їх душі команди ls, grep або make.

GLib – це бібліотека нижнього рівня (lower-level), яка надає багато корисних визначень і функцій, включаючи визначення основних типів і їх обмежень, стандартні макроси, перетворення типів, порядок байтів, розподіл пам'яті, попередження та твердження, журнал повідомлень, таймери, утиліти рядків, функції підключення, лексичний сканер, динамічне завантаження модулів і автоматичне доповнення рядків і ще багато чого іншого.

GLib також визначає низку структур даних (і пов’язаних з ними операцій), зокрема:

- Фрагменти пам'яті
- Двозв'язні списки
- Однозв'язні списки
- Хеш-таблиці

- Рядки (Strings), які можуть динамічно зростати
- Фрагменти (групи) рядків (String chunks)
- Масиви, розмір яких може збільшуватися в міру додавання елементів
- Збалансовані бінарні дерева
- N-арні дерева
- Кварки (двостороння асоціація рядка та унікального цілочисельного ідентифікатора)
- Ключові списки даних (списки елементів даних, доступні за стрінговим або цілим ідентифікатором)
- Відношення та кортежі (таблиці даних, які можна індексувати за будь-якою кількістю полів)
- Кеші (Caches)

Але це не все. При написанні, наприклад, GUI (віконної програми), в якій використовується миша, вам напевно знадобиться цикл обробки подій для перехоплення та диспетчеризації подій миші та клавіатури – у GLib він теж є. До бібліотеки включено також засоби для роботи з файлами, які правильно поводяться в системах, сумісних зі стандартом POSIX та MS Windows. Є також простий аналізатор конфігураційних файлів та полегшений лексичний аналізатор для складніших завдань тощо.

Все вищесказане означає, що напевно є сенс витрачати свій час на освоєння цієї бібліотеки. Мета цієї глави не полягає в тому, щоб бути вичерпним посібником для всієї GLib. Ця бібліотека величезна і розповісти про всі її функції у рамках цієї книги нереально, але про деякі з них ви тут дізнаєтеся.

4.3. Як встановити пакет на Ubuntu

Існує декілька способів встановити відповідне програмне забезпечення. Якщо ви почали користуватися Ubuntu або будь-яким дистрибутивом Linux на базі Ubuntu, таким як Linux Mint тощо, ви напевно натрапили на команду `apt-get` – набір інструментів командного рядка, які дозволяють встановлювати, видаляти та оновлювати пакети deb, встановлені за допомогою APT (Advanced Package Tool) у Debian і Ubuntu. Це полегшує керування пакетами в дистрибутивах на основі Debian. У цій главі посібника використовується Linux Mint, але ви можете використовувати будь-який інший дистрибутив Linux на базі Ubuntu, наприклад, Linux Lite тощо.

Команда `apt-get` в основному працює з базою даних доступних пакетів. Якщо ця база даних не буде оновлена, система не знатиме, чи є доступні новіші пакети. Фактично, це перша команда, яку потрібно запустити в будь-якій системі Linux на базі Debian після нової інсталяції.

Зараз нас цікавить відповідь на питання: як працювати з пакетами, а саме як встановити новий пакет за допомогою `apt-get` і відповідних команд.

Для оновлення бази даних пакетів потрібні права суперкористувача, тому вам знадобиться використовувати `sudo`.

```
$ sudo apt-get update -y
```

Коли ви запустите цю команду, ви побачите, що інформація отримується з різних серверів. Це цілком нормально.

Прапорець `-y` означає відповідь «так» і тихе встановлення, у більшості випадків без запитань.

Після оновлення бази даних пакетів можна оновити встановлені пакети. Найзручніший спосіб – оновити всі пакети, для яких доступні оновлення. Для цього можна використати наведену нижче команду:

```
$ sudo apt-get upgrade
```

Щоб оновити лише певну програму, скористайтеся командою нижче:

```
sudo apt-get upgrade <package_name>
```

Якщо ви знаєте назву пакета, ви можете легко встановити його за допомогою команди:

```
sudo apt-get install <package_name>
```

Для встановлення бібліотеки GLib (libglib2.0-dev) на сервері Ubuntu замініть `<package_name>` на назву пакета.

Все, що потрібно зробити, це використати команду:

```
$ sudo apt-get install -y libglib2.0-dev
```

4.4. Зв'язані списки

Існує чимало бібліотек, якими ви можете скористатися у своїй програмі і при цьому займатися вирішенням саме поставленого завдання, не витрачаючи час на повторну реалізацію зв'язаних списків та інших допоміжних засобів.

Є два типи зв’язаних списків, наданих GLib: однозв'язні та двозв'язні списки: SL-списки і DL-списки (singly linked and doubly linked lists).

GLib надає функції для цих двох типів даних із префіксами `g_slist_foo` та `g_list_foo` відповідно.

Однозв'язні списки (`GSList`) — це найпростіший вид зв’язаного списку, у якому кожен вузол має частину даних і вказівник на наступний вузол. Вказівник на `NULL` позначає останній вузол. Структура `GSList`, яка наведена нижче, представляє один вузол у списку.

```
typedef struct {
  gpointer data;
  GSList *next;
} GSList;
```

Структура `GSList` використовується функціями для отримання довжини, додавання (appending), додавання на початку (prepending), вставки (inserting) та видалення (removing) елементів. Більше інформації про SL-списки і DL-списки можна знайти в оригінальній документації.

Найпростішим контейнером у GLib є однозв'язний список `GSList`. Як випливає з назви, це ряд елементів даних, які пов’язані між собою так, щоб забезпечити перехід від одного елемента даних до іншого. Він називається SL-списком, оскільки між елементами існує лише один зв’язок. Таким чином, можна рухатися лише «вперед» по списку, але немає змоги рухатися вперед, а потім назад у зворотному напрямку.

Щоразу, коли додається елемент до списку, створюється нова структура `GSList`. Ця структура `GSList` складається з елемента даних і вказівника. Попередній кінець списку потім вказує на цей новий вузол, що означає, що тепер новий вузол знаходиться в кінці списку. Термінологія може здатися трохи заплутаною, оскільки вся структура називається `GSList`, і кожен вузол також є структурою `GSList`.

Однак концептуально список – це лише послідовність списків, кожен з яких складається з одного елемента. Це щось на кшталт черги машин на світлофорі; навіть якби на світлофорі стояла лише одна машина, це все одно вважалося б чергою машин.

Пов'язаний список елементів має певні особливості. Визначення довжини списку є операцією O(n). Якщо не порахувати кожний елемент списку, неможливо зрозуміти, наскільки довгий цей список. Додавання на початок списку відбувається швидко (операція O(1)), але пошук елемента – це операція O(n), оскільки потрібно виконати лінійний пошук по всьому списку, доки не буде знайдено потрібний елемент. Додавання елемента в кінець списку також є операцією O(n), оскільки, щоб дістатися до "хвоста", потрібно почати з початку та повторювати до тих пір, доки не буде досягнуто кінця списку.

`GSList` може містити два типи основних даних: цілі чи вказівники. Але насправді це означає, що ви можете додати до `GSList` майже все, що завгодно. Наприклад, якщо вам потрібен `GSList` типу даних short, ви можете просто розмістити вказівники на short у `GSList`.

DL-списки (`GList`) забезпечують ті самі функції, що й SL-списки, за винятком того, що в структурі присутній також вказівник на попередній елемент у списку, що дозволяє переміщатися у будь-якому напрямку. Попередній (`prev`) вказівник на `NULL` позначає, що це – перший елемент у списку.

```
typedef struct {
  gpointer data;
  GList *next;
  GList *prev;
} GList;
```

За винятком можливості проходити DL-список у зворотному напрямку, обидва типи списків забезпечують однакову функціональність. Тому, хоча решту інформації далі буде надано про DL-списки, вона також стосується SL-списків: просто треба змінити префікс функції.

Більшість функцій у наступних прикладах повертають новий вказівник `GList`. Це значення слід зберегти, оскільки розташування початку списку може змінитися через якусь дію, яку виконує функція.

Для того, щоб додати новий елемент на початок списку, використовується функція `g_list_prepend`. Також можна додати елемент на початок списку за допомогою функції `g_list_append`, але цю функцію слід використовувати

розуміючи те, що вона має пройти по списку для того, щоб дістатися до відповідного місця, куди вставити елемент. Можна додавати всі елементи до початку списку, а потім викликати `g_list_reverse`, щоб змінити порядок списку.

```
GList* g_list_prepend (GList *list, gointer data);
```

На додаток до операцій “appending” та “prepending” нових вузлів, можна вставляти вузол у довільну позицію списку за допомогою функції `g_list_insert`. Якщо позиція негативна або більша за кількість вузлів у списку, ця функція діятиме як `g_list_append`. Також можна вставити новий вузол безпосередньо перед іншим за допомогою функції `g_list_insert_before`. Отримати довжину списку можна за допомогою функції `g_list_length`, яка повертає ціле число без знаку.

```
GList* g_list_insert (GList *list, gpointer data, gint position);
```

Можна видалити елемент зі списку за допомогою функції `g_list_remove`, яка видалить перший знайдений вузол зі списку. У той час як `g_list_remove` видаляє лише перше входження вузла, функцію `g_list_remove_all` можна використати для видалення кожного вузла, який має відповідний елемент даних. У разі, якщо відповідного вузла не буде знайдено, зі списком нічого не відбудеться.

```
GList* g_list_remove (GList *list, gconstgpointer data);
```

Якщо треба видалити вузол без звільнення його даних, слід викликати функцію `g_list_remove_link`, яка приймає вказівник на елемент, який треба видалити. Попередній і наступний вказівники мають значення `NULL`, тому вузол стає списком з одного елемента.

По завершенні роботи зі зв’язаними списками, необхідно звільнити пам’ять за допомогою функції `g_list_free`. Зверніть увагу, що звільняється лише сам зв’язаний список, тому потрібно ще буде переконатися в тому, що ви звільнили будь-які динамічно виділені дані, перш ніж викликати цю функцію, інакше це призведе до витоку пам’яті.

```
void g_list_free (GList *list);
```

Сортувати пов’язаний список можна за допомогою функції `g_list_sort`. Для цього потрібно лише вказати `GCompareFunc`. Функції порівняння отримують

два константні вказівники `gconstpointer`. Ці два вказівники посилаються на два вузли, які наразі порівнюються. При порівнянні повертається від'ємне число, якщо перший візол має бути відсортовано перед другим, позитивне число – другий перед першим, і нуль, якщо вони рівні.

```
GList* g_list_sort (GList *list, GCompareFunc compare_func);
```

Існує дві функції для пошуку у зв'язному списку. Стандартною функцією є `g_list_find`, яка знаходить перший елемент у списку з заданими даними. У разі, якщо відповідний вузол не знайдено, ця функція повертає `NULL`.

```
GList* g_list_find (GList *list, gconstpointer data);
```

Також можна задати власну функцію пошуку через `g_list_find_custom`, якщо кожен елемент списку містить складний тип даних. Цей метод використовує той самий формат функції порівняння, що й `g_list_sort`, і повертає відповідний вузол `GList`, коли ви повертаєте 0 із `GCompareFunc`. Ця функція також поверне `NULL`, якщо відповідності не знайдено.

Існує очевидна проблема при використанні зв'язаних списків – виконання багатьох операцій, включаючи сортування, є доволі неефективними під час роботи з довгими списками. Деякі функції вимагають обходу зв'язаного списку, що може зайняти певну кількість часу, коли в списку міститься доволі багато вузлів. Тому їх слід використовувати лише тоді, коли не передбачається дуже велика кількість вузлів.

Ефективне використання зв'язаних списків вимагає уникати додаткових проходжень списку. Одним із можливих рішень є збереження останньої позиції в списку або тих позицій, які планується використовувати в подальшому. Це може скоротити час, необхідний для пошуку певних елементів.

Звізно неможливо повністю уникнути проходження зв'язаного списку. Якщо є потреба виконувати операцію над кожним елементом списку, то слід використовувати функцію `g_list_foreach`, яка буде викликати заданий екземпляр `GFunc` для кожного вузла в списку.

```
void g_list_foreach (GList *list, GFunc func, gpointer data);
```

Прототип `GFunc` приймає елемент даних вузла та параметр даних у `g_list_foreach`. Уникаючи багаторазового проходження зв'язаних списків, їх можна ефективно використовувати для багатьох різних випадків.

GLib має ще багато чого корисного.

4.5. Компіляція і компоновка програм з бібліотекою GLib

Ось проста програма, яка використовує деякі функції бібліотеки GLib:

```
/* File ex4_1.c */
#include <stdio.h>
#include <glib.h>
int main(int argc, char **argv) {
    GList *list = NULL;
    list = g_list_append(list, "Hello world!");
    printf("The first item is '%s'\n", (char *)g_list_first(list)->data);
    g_list_free(list);
    return 0;
}
```

Директиви `#include` вам знайомі. Компілятор вставляє замість них вміст файлів `stdio.h` і `glib.h` відповідно, а отже, і оголошення функцій `printf`, `g_list_append`, `g_list_first`, `g_list_free`, а також визначення типу `GList`.

Тепер займемося питанням щодо того, як зібрати свою програму з бібліотекою GLib.

Типи даних і функції для роботи зі зв'язаними списками, оголошені в заголовку `glib.h`, містяться в окремій бібліотеці GLib, і компонувальнику необхідно сказати про це за допомогою прапорця `-lglib-2.0`. Тут `-l` означає, що необхідно додати бібліотеку, ім'я якої у разі складається з рядка літер `glib-2.0`. До речі, для використання у коді програми функції `printf` додаткових вказівок давати не потрібно, тому що при виклику компілятора неявно мається на увазі, що в самому кінці команди вказаний прапорець `-lc`, що вимагає від компонувальника додати стандартну бібліотеку `libc.` Библиотека GLib 2.0 компонується за допомогою прапорця `-lglib-2.0`.

Таким чином, якщо файл називається `ex4_1.c`, то повна команда виклику компілятора gcc з кількома додатковими прапорцями, які ми обговоримо нижче, виглядає так:

```
$ gcc -Wall -I/usr/include/glib-2.0 ➥
-I/usr/lib/x86_64-linux-gnu/glib-2.0/include ex4_1.c -o ex4_1 -lglib-2.0
```

Отже, у тексті програми за допомогою директиви `#include <glib.h>` компілятору повідомлено про необхідність включити засоби для роботи зі зв'язаними списками, а за допомогою прапорця `-lglib-2.0` у командному рядку компонувальник поінформовано щодо необхідності прикомпонувати бібліотеку GLib. Прапорець `-o` задає ім'я вихідного файлу, без нього за умовчанням отримано буде виконуваний файл `a.out`. Прапорець `-Wall` виводить усі попередження компілятора. Підходить як для компілятора gcc, так і для Clang. Це ще не все. Компілятор повинен знати, в яких каталогах шукати заголовки та об'єктні файли. Пошук цих файлів ускладнюється в тих випадках, коли використовуються бібліотеки, які не описані в стандарті С. У типовій ОС Linux бібліотеки можуть перебувати принаймні в декількох місцях. Розробник операційної системи визначає один або два стандартні каталоги, де знаходяться бібліотеки, що ним поставляються. Що стосується стандартних місць, то тут зазвичай проблем не виникає, компілятор знає, де шукати стандартну бібліотеку С і все встановлене разом з нею, а про все інше компілятору потрібно повідомити явно. І саме ось тут відбувається невеличка магія, бо проблема полягає в тому, що немає ніякого стандартного способу пошуку бібліотек у нестандартних місцях, і ця неприємність викликає роздратування під час компоновки.

Компілятор знає, де знаходяться стандартні каталоги для розташування бібліотек, і тому розповсюджувачі бібліотек намагаються встановлювати їх саме в ці каталоги, щоб користувачам не доводилося вказувати шлях до них вручну.

На цей час на вашому комп'ютері є бібліотека glib-2.0, а ви не знаєте де знаходяться різні файли, що відносяться до неї, тобто щоб сформувати команду компіляції вам необхідно якимось чином визначити шляхи `-I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include`. Тут прапорець `-I` додає шлях до списку каталогів, де компілятор шукає файли, що включаються за допомогою директиви `#include`. Прапорець `-L` додає шлях до списку каталогів, де робиться пошук бібліотек. В нашому конкретному випадку бібліотека glib-2.0 знаходиться в стандартному каталозі, тому ми не використовуємо цей прапорець.

Повернемося до проблеми місцезнаходження: де знаходиться та сама бібліотека, з якою ви хочете скомпонувати програму? Якщо вона була встановлена менеджером пакетів, то, швидше за все, вона знаходиться в одному зі стандартних місць і турбуватися нема про що. Можливо, ви уявляєте, де можуть знаходитися локальні бібліотеки, наприклад у каталозі `/usr/local` або

`/opt`. Зрозуміло, до ваших послуг у ОС є різноманітні засоби пошуку на диску, зокрема визначена у POSIX утиліта `find`. Так, команда

```
find /usr -name 'glib-2.0*'
```

шукає в каталозі `/usr` файли з іменами, що починаються з `glib-2.0`.

```
$ find /usr -name 'glib-2.0*'
/usr/share/aclocal/glib-2.0.m4
/usr/share/glib-2.0
/usr/lib/i386-linux-gnu/glib-2.0
/usr/lib/x86_64-linux-gnu/pkgconfig/glib-2.0.pc
/usr/lib/x86_64-linux-gnu/glib-2.0
/usr/include/glib-2.0

$ ls -l /usr/include/glib-2.0
total 44
drwxr-xr-x 2 root root 12288 May 30 20:22 gio
drwxr-xr-x 3 root root  4096 May 30 20:22 glib
-rw-r--r-- 1 root root  3437 Sep 21  2023 glib.h
-rw-r--r-- 1 root root  1531 Sep 21  2023 glib-object.h
-rw-r--r-- 1 root root  4461 Sep 21  2023 glib-unix.h
-rw-r--r-- 1 root root  5080 Sep 21  2023 gmodule.h
drwxr-xr-x 2 root root  4096 May 30 20:22 gobject
```

Але полювання за бібліотеками по всьому диску мало кому подобається, тому існує утиліта `pkg-config`, яка вирішує цю проблему, зберігаючи в репозиторії прапорці та місця, які пакети оголошують необхідними для компіляції. Наберіть у командному рядку `pkg-config`; якщо з'явилося повідомлення з проханням надати імена пакетів, значить все нормально: у вас є `pkg-config`, і нею можна скористатися для подальших досліджень.

Я виконую наступні команди своєму комп'ютері:

```
$ pkg-config --cflags glib-2.0
-I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include

$ pkg-config --libs glib-2.0
-lglib-2.0
```

Результати виконання цих команд – є ті самі прапорці, які потрібно використати в команді компіляції. Бачимо, що бібліотека GLib перебуває у стандартному місці, бо прапорців `-L` немає.

Команда `pkg-config` робить багато корисного, але в стандарті вона не описана, тому не можна очікувати, що вона є на будь-якій машині або будь-яка бібліотека реєструється в її репозиторії. Якщо у вас цієї утиліти немає, доведеться зайнятися самостійними дослідженнями: прочитати посібник з бібліотеки або пошукати на диску, як було показано вище.

Зауважте, що тепер більше не потрібно вказувати шляхи до файлів заголовків GLib; опція `pkg-config --cflags` подбає про це. Те саме стосується бібліотек, на які вказує параметр `--libs`.

Тут немає ніякої магії; `pkg-config` повертає розташування файлів заголовків із файлу конфігурації. У системі Linux Mint файли `pkg-config` знаходяться в `/usr/lib/x86_64-linux-gnu/pkgconfig`, а файл `glib-2.0.pc` виглядає ось так:

```
$ cat -n /usr/lib/x86_64-linux-gnu/pkgconfig/glib-2.0.pc
     1 prefix=/usr
     2 libdir=${prefix}/lib/x86_64-linux-gnu
     3 includedir=${prefix}/include
     4
     5 bindir=${prefix}/bin
     6 glib_genmarshal=${bindir}/glib-genmarshal
     7 gobject_query=${bindir}/gobject-query
     8 glib_mkenums=${bindir}/glib-mkenums
     9
    10 Name: GLib
    11 Description: C Utility Library
    12 Version: 2.72.4
    13 Requires.private: libpcre >=  8.31
    14 Libs: -L${libdir} -lglib-2.0
    15 Libs.private: -pthread -lm
    16 Cflags: -I${includedir}/glib-2.0 -I${libdir}/glib-2.0/include
```

До речі, згадаємо дещо, про що йшлося в розділі "Зв’язані списки":

```
$ cat -n /usr/include/glib-2.0/glib/glist.h
. . .
    37 typedef struct _GList GList;
    38
```

```
    39 struct _GList
    40 {
    41  gpointer data;
    42  GList *next;
    43  GList *prev;
    44 };
. . .
```

Ви бачите, що `gpointer` – загальний нетипізований pointer, який визначається як `void*`. Він призначений для того, щоб виглядати привабливіше, ніж стандартний тип `void*`

А тепер давайте все те, про що йшлося вище, зберемо до купи:

```
$ pkg-config --cflags glib-2.0
-I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include

$ pkg-config --libs glib-2.0
-lglib-2.0

$ gcc -Wall -I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include
ex4_1.c -o ex4_1 -lglib-2.0

$ ./ex4_1
The first item is 'Hello world!'
```

Скористаємося отриманими знаннями для створення і компіляції ще однієї програми, яка використовує бібліотеку glib-2.0:

```
/* File simplelist.c */
#include <stdio.h>
#include <glib.h>

int main() {
    GList *list=NULL;
    list = g_list_append(list, "a");
    list = g_list_append(list, "b");
    list = g_list_append(list, "c");

    for ( ; list != NULL; list=list->next) {
        printf("%s\n", (char *)list->data);
    }
```

```
    g_list_free(list);
    return 0;
}
```

```
$ gcc -Wall -I/usr/include/glib-2.0 ➥
-I/usr/lib/x86_64-linux-gnu/glib-2.0/include simplelist.c ➥
-o simplelist -lglib-2.0

$ ./simplelist
a
b
c
```

Все працює, але команда компіляції виглядає доволі довгою і користуватися такою командою незручно. Навіть за наявності корисної утиліти `pkg-config`, виникає гостра потреба у якомусь інструменті, який автоматично збирає все необхідне до купи. Кожен окремий елемент зрозуміти нескладно, але довга механічна робота втомлює. Рішення є: існує спеціальна утиліта make, яка обробляє організований набір змінних та однорядкових скриптів оболонки. Вона описана у стандарті POSIX і читає команди та змінні із спеціального файла, який називається makefile. Якщо ви добре зрозуміли те, про що йшлося у цьому розділі, то ви навряд чи станете коли-небудь в таких умовах викликати компілятор безпосередньо і використовувати довгі командні рядки, а скористаєтеся makefile.

4.6. Декілька слів про Makefile

Програму `simplelist.c` було раніше скомпільовано за допомогою команди:

```
gcc -Wall -I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include ➥
simplelist.c -o simplelist -lglib-2.0
```

Без використання make-файлу типовим підходом до циклу тестування/зміни/налагодження є використання стрілки вгору в терміналі, щоб повернутися до вашої останньої команди компіляції. Завдяки цьому вам не потрібно вводити команду щоразу. На жаль, цей підхід до компіляції має два недоліки. По-перше, якщо ви втратите команду компіляції або перейдете до іншого комп'ютера, вам доведеться повторно вводити цю довгу команду з нуля,

що в кращому випадку неефективно. По-друге, вище було розглянуто елементарні програми: код кожної з них розташовано було лише в одному файлі, а реальні проекти складаються з багатьох файлів, і якщо ви вносите зміни лише в один файл .c, повторна компіляція всіх їх щоразу також забирає багато часу та є неефективною. Отже, настав час подивитися, що ми можемо зробити за допомогою використання make-файла.

GNU Make – це інструмент, який керує створенням виконуваних файлів та інших файлів з вихідних файлів програми. Для цього система збірки Make використовує файли Makefile. Makefile – текстовий файл (без будь-якого розширення), який знаходиться в каталозі з вихідними файлами і містить цілі збірки та команди, що дозволяють Make зрозуміти те, як зібрати поточну кодову базу.

Відомості про те, як побудувати користувацьку програму, Make отримує з файлу під назвою makefile, у якому вказані необхідні інструкції. Коли ви створюєте програму, ви маєте написати для неї відповідний makefile. Це надасть змогу зробити процеси компіляції, компонування та встановлення вашої програми більш зручними. Цій темі присвячено багато книг та статей в інтернеті. Основне джерело інформації https://www.gnu.org/software/make/

Make-файли – це простий спосіб організації компіляції коду. Тут не йдеться про все те, що можна зробити за допомогою make, бо цей розділ призначений лише для початківців, щоб надати змогу швидко та легко створювати власні прості make-файли.

Найпростіший make-файл, який ви можете створити, виглядатиме приблизно так:

```
$ cat -n Makefile
     1 CC=gcc
     2 CFLAGS=-Wall -I/usr/include/glib-2.0 ➥
-I/usr/lib/x86_64-linux-gnu/glib-2.0/include
     3 LDFLAGS=-lglib-2.0
     4
     5 simplelist: simplelist.o
     6     $(CC) -o simplelist simplelist.o $(LDFLAGS)
```

Створіть такий файл, назвіть його Makefile або makefile. Номери рядків вказувати не потрібно. Введіль команду make у командному рядку. В результаті

буде виконано команду компіляції, яка записана у makefile (рядок 6). Зауважте, що make без аргументів виконує перше правило у файлі (рядок 5). Крім того, розміщуючи список файлів, від яких залежить команда, у рядку після `:`, ви повідомляєте make про те, що правило `simplelist` (рядок 5) потрібно виконати, якщо будь-який із переліку файлів змінюється (у нашому випадку це стосується лише файла `simplelist.c`). Цим було вирішено проблему №1 і тим самим тепер можна уникнути необхідності повторного використання клавіші стрілки вгору, шукаючи останню команду компіляції. Важливо зауважити, що перед командою `$(CC)` у make-файлі є символ табуляції (рядок 6). На початку будь-якої команди у make-файлі має бути символ табуляції!

В оболонці shell і в програмі make значення змінної позначається знаком `$`, але в оболонці пишуть `$var`, а в make ім'я змінної, яке довше одного символу, слід писати у дужках: `$(var)`.

Як було показано вище, в makefile, можна задати змінну на початку файла, в рядках виду `CFLAGS=...` . У makefile дозволяється залишати пробіли до і після знаку `=`. Тут змінна `$(CC)`представляє компілятор; у стандарті POSIX визначено, що за умовчанням `CC=c99`, але в сучасних версіях GNU make вважається, що `CC=cc`, а це зазвичай є посиланням на gcc.

Також було використано ще деякі константи: `CC`, `CFLAGS` і `LDFLAGS`. Це спеціальні константи, які потрібні для того, щоб визначити те, як скомпілювати файл `simplelist.c`. Зокрема, макрос `CC` – це компілятор C, який потрібно використовувати, а `CFLAGS` – це список прапорців, які потрібно передати команді компіляції.

У `CFLAGS` і `LDFLAGS` вказуються прапорці, які необхідні компілятору, та для компонування бібліотеки. Якщо у системі є програма `pkg-config`, то можна скористайтеся варантом зі зворотними апострофами:

```
CFLAGS=`pkg-config --cflags glib-2.0`
LDFLAGS=`pkg-config --libs glib-2.0`
```

Завдяки розміщенню об’єктного файлу `simplelist.o` у списку залежностей (рядок 5) і в самому правилі (рядок 6), make зрозуміє, що спочатку потрібно окремо скомпілювати `simplelist.c`, а потім зібрати виконуваний файл `simplelist`.

```
$ make simplelist
gcc -Wall -I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include -c
-o simplelist.o simplelist.c
```

```
gcc -o simplelist simplelist.o -lglib-2.0
$ ./simplelist
a
b
c
```

Make-файл можна використовувати для формування правил компіляції декількох файлів, наприклад `simplelist.c` і `ex4_1.c`:

```
     1 CC=gcc
     2 CFLAGS=-Wall -I/usr/include/glib-2.0 ➥
-I/usr/lib/x86_64-linux-gnu/glib-2.0/include
     3 LDFLAGS=-lglib-2.0
     4
     5 simplelist: simplelist.o
     6     $(CC) -o simplelist simplelist.o $(LDFLAGS)
     7
     8 ex4_1: ex4_1.o
     9     $(CC) -o $@ $^ $(LDFLAGS)
```

До того ж для спрощення можна використати спеціальні макроси `$@` та `$^`, щоб зробити правило компіляції більш загальним. У make є також кілька вбудованих змінних. Наприклад, `$@` - повне ім'я цільового файлу. Під цільовим розуміється файл, який належить створити, наприклад о-файл, що утворюється у результаті компіляції с-файла, чи програма, що є результатом виконання компонування о-файлів.

```
$ ls -l
total 12
-rw-rw-r-- 1 vadim vadim 393 May 30 23:07 ex4_1.c
-rw-rw-r-- 1 vadim vadim 280 Jun  7 14:32 Makefile
-rw-rw-r-- 1 vadim vadim 319 Jun  7 13:23 simplelist.c

$ cat -n Makefile
     1 CC=gcc
     2 CFLAGS=-Wall -I/usr/include/glib-2.0
-I/usr/lib/x86_64-linux-gnu/glib-2.0/include
     3 LDFLAGS=-lglib-2.0
     4
     5 simplelist: simplelist.o
     6     $(CC) -o simplelist simplelist.o $(LDFLAGS)
     7
```

```
 8 ex4_1: ex4_1.o
 9     $(CC) -o $@ $^ $(LDFLAGS)
10
11 clean:
12     rm -rf *.o simplelist ex4_1
```

```
$ make ex4_1
gcc -Wall -I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include -c
-o ex4_1.o ex4_1.c
gcc -o ex4_1 ex4_1.o -lglib-2.0

$ ./ex4_1
The first item is 'Hello world!'

$ make simplelist
gcc -Wall -I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include -c
-o simplelist.o simplelist.c
gcc -o simplelist simplelist.o -lglib-2.0

$ ./simplelist
a
b
C
```

Тема створення і використання Makefile заслуговує окремого детального розглядання. Тут було розглянуто лише невеликий шматок в якості ліричного відступу від головної теми, який надає можливість частково спростити процес розробки.

4.7. Однозв'язний список

Найпростішим контейнером у GLib мабуть можна вважати однозв'язний список – `GSList`. Як випливає з назви, це послідовність елементів даних, які пов’язані між собою, завдяки чому забезпечується можливість переходити від одного елемента даних до іншого. Він називається однозв'язним списком, оскільки між елементами існує лише один (односпрямований) зв’язок. Таким чином, можна рухатися лише «вперед» по списку, але при цьому неможливо рухатися назад по списку у зворотному напрямку.

4.7.1. Створення, додавання та видалення

Наступний код ініціалізує `GSList`, додає до нього два елементи, друкує довжину списку та звільняє ресурс:

```
/* File ex4_2.c */
#include <stdio.h>
#include <glib.h>
int main() {
    GSList *list = NULL;
    printf("The list is now %d items long\n", g_slist_length(list));
    list = g_slist_append(list, "first");
    list = g_slist_append(list, "second");
    printf("The list is now %d items long\n", g_slist_length(list));
    g_slist_free(list);
}
```

Результат виконання:

```
The list is now 0 items long
The list is now 2 items long
```

Звеніть увагу ось на що. Більшість функцій GLib мають формат g_(назва контейнера)_(назва функції). Отже, щоб отримати довжину `GSList`, ви повинні викликати функцію `g_slist_length`.

Для створення нового `GSList` окремої функції немає. Замість цього просто оголошується вказівник на структуру `GSList` і призначається йому значення `NULL`.

Функція `g_slist_length` повертає довжину списка. Вона також може повернути довжину порожнього списку; у цьому випадку вона повертає 0.

Функція `g_slist_append` повертає вказівник на новий початок списку, тому потрібно зберегти це значення.

Функція `g_slist_free` звільняє пам’ять, тобто видаляє елементи, розміщені в списку.

4.7.2. Додавання та видалення елементів

Ви можете як додавати дані до списку, так і видаляти їх:

```
/* File ex4_3.c */
#include <stdio.h>
#include <glib.h>
int main() {
    GSList *list = NULL;
    list = g_slist_append(list, "second");
    list = g_slist_prepend(list, "first");
    printf("The list is now %d items long\n", g_slist_length(list));
    list = g_slist_remove(list, "first");
    printf("The list is now %d items long\n", g_slist_length(list));
    g_slist_free(list);
}
```

Результат виконання:

```
The list is now 2 items long
The list is now 1 items long
```

Більшість цього коду виглядає знайомим, але є деякі моменти, на які слід звернути увагу:

Якщо ви викликаєте функцію `g_slist_remove` та передаєте їй елемент, якого немає у списку, то список не зміниться. Функція `g_slist_remove` також повертає новий початок списку.

Ви бачите, що `"first"` додається за допомогою виклику функції `g_slist_prepend` до початку списка. Звісно, це більш швидкий виклик, ніж функції `g_slist_append`, бо це O(1), а не O(n), тому що, як згадувалося раніше, виконання додавання `g_slist_append` вимагає повного обходу списку.

4.7.3. Видалення повторюваних елементів

Ось приклад, в якому створюється декілька однакових елементів (дублікатів) у списку:

```
/* File ex4_4.c */
#include <stdio.h>
#include <glib.h>
int main() {
    GSList *list = NULL;
    list = g_slist_append(list, "first");
```

```
    list = g_slist_append(list, "second");
    list = g_slist_append(list, "second");
    list = g_slist_append(list, "third");
    list = g_slist_append(list, "third");
    printf("The list is now %d items long\n", g_slist_length(list));
    list = g_slist_remove(list, "second");
    list = g_slist_remove_all(list, "third");
    printf("The list is now %d items long\n", g_slist_length(list));
    g_slist_free(list);
}
```

Результат виконання:

```
The list is now 5 items long
The list is now 2 items long
```

Якщо `GSList` містить один і той самий елемент двічі, а ви викликаєте функцію `g_slist_remove`, то буде видалено лише перший знайдений елемент, але можна видалити всі входження елемента за допомогою функції `g_slist_remove_all`.

4.7.4. Вибірка довільного і наступного елемента

Після того, як було додано кілька елементів у список `GSList`, є можливість вибирати елементи різними способами. Ось кілька прикладів:

```
/* File ex4_5.c */
#include <stdio.h>
#include <glib.h>
int main() {
    GSList *list = NULL;
    list = g_slist_append(list, "first");
    list = g_slist_append(list, "second");
    list = g_slist_append(list, "third");
    printf("The last item is '%s'\n",
         (char *)g_slist_last(list)->data);
    printf("The item at index '1' is '%s'\n",
         (char *)g_slist_nth(list, 1)->data);
    printf("Now the item at index '1' the easy way: '%s'\n",
         (char *)g_slist_nth_data(list, 1));
    printf("And the 'next' item after first item is '%s'\n",
```

```
          (char *)g_slist_next(list)->data);
    g_slist_free(list);
}
```

Результат виконання:

```
The last item is 'third'
The item at index '1' is 'second'
Now the item at index '1' the easy way: 'second'
And the 'next' item after first item is 'second'
```

Зверніть увагу на те, що є деякі додаткові функції на `GSList`: порівняйте використання функцій `g_slist_nth_data` і `g_slist_nth`.

`g_slist_next` – це не не зовсім виклик функції, а радше макрос. У результаті макроподстановки він замінюється посиланням на наступний елемент у `GSList`. В коді було передано перший елемент `GSList`, тому макрос створить посиляння на другий елемент. Це швидка операція, оскільки немає додаткових витрат на виклик функції.

4.7.5. Робота з користувацьким типом

До цього моменту ми працювали лише з рядками символів в `GSList`. У наведеному нижче прикладі визначено структуру даних `Person` і кілька її екземплярів додано у `GSList`:

```
/* File ex4_6.c */
#include <stdio.h>
#include <glib.h>
typedef struct {
    char *name;
    int age;
} Person;
int main() {
    GSList *list = NULL;
    Person *p1 = malloc(sizeof(Person));
    p1->name = "John";
    p1->age = 32;
    list = g_slist_append(list, p1);
    Person *p2 = g_new(Person, 1);
```

```
    p2->name = "Sam";
    p2->age = 21;
    list = g_slist_append(list, p2);
    printf("Person 1 age is '%d'\n", ((Person *)list->data)->age);
    printf("The last Person's name is '%s'\n",
         ((Person *)g_slist_last(list)->data)->name);
    g_slist_free(list);
    free(p1);
    g_free(p2);
}
```

Результат виконання:

```
Person 1 age is '32'
The last Person's name is 'Sam'
```

Кілька слів щодо роботи з GLib і типами, визначеними користувачем.

Розміщення визначеного користувачем типу в `GSList` є таким самим, як рядок символів. Зверніть увагу на те, що потрібно зробити приведення типу, коли ви отримуєте елемент зі списку.

У цьому прикладі використовується інший макрос GLib – макрос `g_new` - для створення екземпляра `Person p2`. Цей макрос використовується, щоб викликати `malloc` для виділення необхідного об'єму пам’яті для заданого типу. Це трохи зручніше, ніж самотужки викликати функцію `malloc`.

Нарешті, якщо було виділено пам'ять, то необхідно її звільнити. У прикладі використовується функція GLib `g_free`, яка звільняє пам'ять екземпляра `Person p2` (оскільки йому було виділено пам'ять за допомогою `g_new`). У більшості випадків `g_free` просто огортає звичайну функцію `free`.

4.7.6. Об’єднання та реверсування списків

GSList має деякі зручні допоміжні функції, які можуть об’єднувати (concatenate) списки та змінювати порядок елементів у списку на зворотній (reverse). Ось як вони працюють:

```
/* File ex4_7.c */
#include <stdio.h>
#include <glib.h>
```

```
int main() {
    GSList *list1 = NULL;
    list1 = g_slist_append(list1, "first");
    list1 = g_slist_append(list1, "second");
    GSList *list2 = NULL;
    list2 = g_slist_append(list2, "third");
    list2 = g_slist_append(list2, "fourth");
    GSList *both = g_slist_concat(list1, list2);
    printf("The third item in the concatenated list is '%s'\n",
         (char *)g_slist_nth_data(both, 2));
    GSList *reversed = g_slist_reverse(both);
    printf("The first item in the reversed list is '%s'\n",
         (char *)reversed->data);
    g_slist_free(reversed);
}
```

Результат виконання:

```
The third item in the concatenated list is 'third'
The first item in the reversed list is 'fourth'
```

Як і очікувалося, два списки були об’єднані в один список, так що перший елемент у `list2` став третім у новому списку. Зауважте, що елементи не копіюються, тому пам'ять потрібно звільнити лише один раз.

Крім того, ви можете роздрукувати перший елемент у перевернутому списку, використовуючи лише розіменування вказівника (`reversed->data`). Оскільки кожен елемент у `GSList` є вказівником на структуру `GSList`, вам не потрібно викликати функцію, щоб отримати перший елемент.

4.7.7. Проста ітерація

Ось простий спосіб перегляду вмісту `GSList`:

```
/* File ex4_8.c */
#include <stdio.h>
#include <glib.h>
int main() {
    GSList *list = NULL, *iterator = NULL;
    list = g_slist_append(list, "first");
    list = g_slist_append(list, "second");
```

```
    list = g_slist_append(list, "third");
    for (iterator = list; iterator; iterator = iterator->next) {
        printf("Current item is '%s'\n", (char *)iterator->data);
    }
    g_slist_free(list);
}
```

Результат виконання:

```
Current item is 'first'
Current item is 'second'
Current item is 'third'
```

Об’єкт ітератора – це просто змінна, оголошена як вказівник на структуру `GSList`. Оскільки однозв'язний список є серією структур `GSList`, ітератор і список мають бути одного типу.

Зауважте також, що цей приклад використовує загальну ідіому GLib: він оголошує змінну ітератора одночасно з оголошенням самого `GSList`.

Нарешті, вираз виходу з циклу `for` перевіряє, чи є ітератор `NULL`. Це спрацьовує, оскільки ітератор буде `NULL` лише після того, як цикл пройде останній елемент у списку.

4.7.8. Розширена ітерація з функціями

Інший спосіб ітерації по `GSList` – це використання функції `g_slist_foreach` і надання в якості її (другого) аргументу імені функції, яка має виконуватися для кожного елемента в списку.

```
/* File ex3_9.c */
#include <stdio.h>
#include <glib.h>
void print_iterator(gpointer item, gpointer prefix) {
    printf("%s %s\n", (char *)prefix, (char *)item);
}
void print_iterator_short(gpointer item) {
    printf("%s\n", (char *)item);
}
int main() {
    GSList *list = g_slist_append(NULL, g_strdup("first"));
    list = g_slist_append(list, g_strdup("second"));
```

```
    list = g_slist_append(list, g_strdup("third"));
    printf("Iterating with a function:\n");
    g_slist_foreach(list, print_iterator, "-->");
    printf("Iterating with a shorter function:\n");
    g_slist_foreach(list, (GFunc)print_iterator_short, NULL);
    printf("Now freeing each item\n");
    g_slist_foreach(list, (GFunc)g_free, NULL);
    g_slist_free(list);
}
```

Результат виконання:

```
Iterating with a function:
--> first
--> second
--> third
Iterating with a shorter function:
first
second
third
Now freeing each item
```

Оператор на зразок

```
GSList x = g_slist_append(NULL, [шось_таке])
```

дозволяє оголосити, ініціалізувати та додати перший елемент до списку одним махом. Функція `g_strdup` зручна для дублювання рядка: просто не забудьте звільнити його, коли закінчите з ним.

Функція `g_slist_foreach` дозволяє передати вказівник, тож ви можете ефективно надати їй будь-який аргумент разом із кожним елементом у списку. Наприклад, ви можете передати накопичувач і збирати інформацію про кожен елемент у списку. Єдине обмеження на функцію ітерації полягає в тому, що вона приймає принаймні один `gpointer` як аргумент. Зверніть увагу на те, як використовується функція `print_interator_short`.

У коді звільняється пам’ять за допомогою вбудованої функції GLib в якості аргументу `g_slist_foreach`. Усе, що вам потрібно зробити в цьому випадку, це використати функцію `g_free` у `GFunc`. Однак вам все одно потрібно звільнити сам `GSList` за допомогою окремого виклику `g_slist_free`.

4.7.9. Сортування за допомогою GCompareFunc

Існує можливість відсортувати `GSList`, надавши функцію, яка знає, як порівнювати елементи списку. У наступному прикладі показано один із способів сортування списку рядків:

```
/* File ex4_10.c */
#include <stdio.h>
#include <glib.h>
gint my_comparator(gconstpointer item1, gconstpointer item2) {
 return g_ascii_strcasecmp(item1, item2);
}
int main() {
 GSList* list = g_slist_append(NULL, "Kyiv");
 list = g_slist_append(list, "Vinnytsia");
 list = g_slist_append(list, "Odesa");
 list = g_slist_sort(list, (GCompareFunc)my_comparator);
 printf("The first item is now '%s'\n", (char *)list->data);
 printf("The last item is now '%s'\n", (char *)g_slist_last(list)->data);
 g_slist_free(list);
}
```

Результат виконання:

```
The first item is now 'Kyiv'
The last item is now 'Vinnytsia'
```

`GCompareFunc` повертає від'ємне значення (negative value), якщо перший елемент менший за другий, 0, якщо вони рівні, і додатне значення (positive value), якщо другий елемент більший за перший. Якщо функція порівняння відповідає цій специфікації, вона може робити в середині свого тіла все те, що їй потрібно.

Оскільки деякі функції GLib слідують цьому шаблону, то їх теж можна використовувати. Фактично, у наведеному вище прикладі ви можете так само легко замінити виклик `my_comparator` на щось на кшталт:

```
g_slist_sort(list, (GCompareFunc)g_ascii_strcasecmp)
```

і ви отримаєте ті самі результати.

4.7.10. Пошук елемента

Існує кілька способів знайти потрібний елемент у `GSList`. Це можна зробити послідовно переглядаючи вміст списку, порівнюючи кожен елемент, доки не знайдеться цільовий елемент, або можна безпосередньо використовувати функцію `g_slist_find` для пошуку потрібного елемента в списку. Нарешті, ви можете використовувати функцію `g_slist_find_custom`, яка дозволяє задати функцію для перевірки кожного елемента в списку:

```
/* File ex4_11.c */
#include <stdio.h>
#include <glib.h>
gint my_finder(gconstpointer item) {
 return g_ascii_strcasecmp(item, "second");
}
int main() {
    GSList *list = g_slist_append(NULL, "first");
    list = g_slist_append(list, "second");
    list = g_slist_append(list, "third");
    GSList *item = g_slist_find(list, "second");
    printf("This should be the 'second' item: '%s'\n", (char *)item->data);
    item = g_slist_find_custom(list, NULL, (GCompareFunc)my_finder);
    printf("Again, this should be the 'second' item: '%s'\n", (char
*)item->data);
    item = g_slist_find(list, "delta");
    printf("'delta' is not in the list, so we get: '%s'\n",
         item ? (char *)item->data : "(null)");
    g_slist_free(list);
}
```

Результат виконання:

```
This should be the 'second' item: 'second'
Again, this should be the 'second' item: 'second'
'delta' is not in the list, so we get: '(null)'
```

Функція `g_slist_find_custom` також приймає у другому параметрі вказівник, тому за потреби можна передати їй додаткову інформацію для того, щоб допомогти функції пошуку. Функція `GCompare` є останнім параметром, а не

другим, оскільки вона знаходиться в `g_slist_sort`. Нарешті, невдалий пошук повертає `NULL`.

4.7.11. Розширене додавання зі вставкою

Тепер, після прикладів використання `GCompareFunc`, є сенс розглянути деякі з операцій вставки. Елементи можна вставляти в задану позицію списку, перед вказаним елементом за допомогою функції `g_slist_insert_before`, а також з використанням порядку сортування за допомогою функції `g_slist_insert_sorted`. Ось як це виглядає:

```
/* File ex4_12.c */
#include <stdio.h>
#include <glib.h>
int main() {
    GSList *list = g_slist_append(NULL, "Red ");
    list = g_slist_append(list, "Orange ");
    printf("Before inserting 'Yellow', second item is: '%s'\n",
         (char *)g_slist_nth(list, 1)->data);
    list = g_slist_insert(list, "Yellow ", 1);
    printf("After insertion, second item is: '%s'\n",
         (char *)g_slist_nth(list, 1)->data);
    list = g_slist_insert_before(list, g_slist_nth(list, 2), "Green ");
    printf("After an insert_before, third item is: '%s'\n",
         (char *)g_slist_nth(list, 2)->data);
    list = g_slist_insert_sorted(list, "Blue ",
         (GCompareFunc)g_ascii_strcasecmp);
    printf("After inserting 'Blue', here's the final list:\n");
    g_slist_foreach(list, (GFunc)printf, NULL);
    putchar('\n');
    g_slist_free(list);
}
```

Результат виконання:

```
Before inserting 'Yellow', second item is: 'Orange '
After insertion, second item is: 'Yellow '
After an insert_before, third item is: 'Green '
After inserting 'Blue', here's the final list:
```

```
Blue Red Yellow Green Orange
```

Оскільки функція `g_slist_insert_sorted` приймає `GCompareFunc`, то можна повторно використовувати вбудовану функцію GLib `g_ascii_strcasecmp`. Тепер має бути зрозуміло, чому наприкінці кожного елемента є додатковий пробіл; це зроблено для того, щоб наприкінці прикладу коду міг з'явитися ще один приклад `g_slist_foreach`, цього разу з `(GFunc)printf`.

4.8. Двозв'язний список

Двозв'язні списки дуже схожі на списки з одним зв’язком, але вони містять додаткові вказівники, щоб отримати більше можливостей для навігації. Маючи вузол у подвійно зв'язаному списку, можна рухатися вперед або назад. Це робить їх більш гнучкими, ніж однозв'язні списки, але також збільшує використання пам’яті, тому недоцільно використовувати подвійний зв’язаний список, якщо справді не потрібна така гнучкість.

GLib містить реалізацію двозв’язаного списку, яка називається `GList`. Більшість операцій у `GList` подібні до операцій у `GSList`. Ми розглянемо кілька прикладів базового використання, а потім додаткові операції, які дозволяє `GList`.

4.8.1. Основні операції над двозв’язними списками

Ось деякі з типових операцій, які можна виконувати з GList:

```
/* File ex4_13.c */
#include <stdio.h>
#include <glib.h>
int main() {
    GList *list = NULL;
    list = g_list_append(list, "Red ");
    printf("The first item is '%s'\n", (char *)list->data);
    list = g_list_insert(list, "Green ", 1);
    printf("The second item is '%s'\n", (char *)g_list_next(list)->data);
    list = g_list_remove(list, "Green ");
    printf("After removal of 'Green', the list length is %d\n",
         g_list_length(list));
    GList *other_list = g_list_append(NULL, "Green ");
    list = g_list_concat(list, other_list);
```

```
    printf("After concatenation: ");
    g_list_foreach(list, (GFunc)printf, NULL);
    list = g_list_reverse(list);
    printf("\nAfter reversal: ");
    g_list_foreach(list, (GFunc)printf, NULL);
    putchar('\n');
    g_list_free(list);
}
```

Результат виконання:

```
The first item is 'Red '
The second item is 'Green '
After removal of 'Green', the list length is 1
After concatenation: Red Green
After reversal: Green Red
```

Наведений вище код, ймовірно, виглядає досить знайомим. Усі перераховані вище операції також присутні в `GSList`. Єдина відмінність для `GList` полягає в тому, що назви функцій починаються з `g_list`, а не з `g_slist`. І, звичайно, всі вони приймають вказівник на структуру `GList`, а не вказівник на структуру `GSList`.

Тепер, коли ви побачили деякі основні операції `GList`, ось кілька операцій, які стають можливими завдяки тому, що кожен вузол у `GList` має посилання на попередній вузол:

```
/* File ex4_14.c */
#include <stdio.h>
#include <glib.h>
int main() {
    GList *list = g_list_append(NULL, "Blue ");
    list = g_list_append(list, "Yellow ");
    list = g_list_append(list, "Black ");
    printf("Here's the list: ");
    g_list_foreach(list, (GFunc)printf, NULL);
    GList *last = g_list_last(list);
    printf("\nThe first item (using g_list_first) is '%s'\n",
         (char *)g_list_first(last)->data);
    printf("The next-to-last item is '%s'\n",
         (char *)g_list_previous(last)->data);
    printf("The next-to-last item is '%s'\n",
         (char *)g_list_nth_prev(last, 1)->data);
```

```
    g_list_free(list);
}
```

Результат виконання:

```
Here's the list: Blue Yellow Black
The first item (using g_list_first) is 'Blue '
The next-to-last item is 'Yellow '
The next-to-last item is 'Yellow '
```

Нічого надто дивного тут не має бути, але потрібно зробити кілька зауважень.

Другий аргумент функції `g_list_nth_prev` є цілим числом, яке вказує, наскільки далеко назад ви хочете перейти. Якщо ви передаєте значення, яке виходить за межі `GList`, готуйтеся до збою.

Функція `g_list_first` є операцією O(n): вона починається з указаного вузла та просувається у зворотному напрямку по списку, поки не знайде початок `GList`. Наведений вище приклад є найгіршим сценарієм, оскільки обхід починається в кінці списку. Функція `g_list_last` має O(n) з тієї ж причини.

4.8.2. Видалення вузлів за допомогою посилань

Ви вже бачили, як можна видалити вузол зі списку, якщо у вас є вказівник на дані, які він містить. Функція `g_list_remove` якраз це і робить. Якщо у вас є вказівник на сам вузол, ви можете видалити цей вузол безпосередньо. Ця операція швидка O(1):

```
/* File ex4_15.c */
#include <stdio.h>
#include <glib.h>
int main() {
    GList *list = g_list_append(NULL, "Red ");
    list = g_list_append(list, "Blue ");
    list = g_list_append(list, "Green ");
    printf("Here's the list: ");
    g_list_foreach(list, (GFunc)printf, NULL);
    GList *blue = g_list_nth(list, 1);
    list = g_list_remove_link(list, blue);
    g_list_free_1(blue);
```

```
    printf("\nHere's the list after the remove_link call: ");
    g_list_foreach(list, (GFunc)printf, NULL);
    list = g_list_delete_link(list, g_list_nth(list, 1));
    printf("\nHere's the list after the delete_link call: ");
    g_list_foreach(list, (GFunc)printf, NULL);
    putchar('\n');
    g_list_free(list);
}
```

Результат виконання:

```
Here's the list: Red Blue Green
Here's the list after the remove_link call: Red Green
Here's the list after the delete_link call: Red
```

Отже, якщо у вас є вказівник на вузол, а не на дані вузла, ви можете видалити цей вузол за допомогою функції `g_list_remove_link`.

Після видалення вам потрібно буде явно звільнити його за допомогою `g_list_free_1`, який робить те, що випливає з назви: звільняє один вузол. Як завжди, вам потрібно зберегти значення, яке повертає `g_list_remove_link`, оскільки це новий початок списку.

Нарешті, якщо все, що ви хочете зробити, це видалити вузол і звільнити його, ви можете зробити це одним кроком за допомогою виклику функції `g_list_delete_link`.

Такі ж функції існують і для `GSList`; просто замініть `g_list` на `g_slist`, і вся наведена вище інформація буде застосована.

4.8.3. Індекси та позиції

Якщо ви просто хочете знайти позицію елемента в `GList`, у вас є два варіанти. Ви можете використовувати виклик `g_list_index`, який шукає елемент, використовуючи дані в ньому, або ви можете використовувати `g_list_position`, який використовує вказівник на вузол. Наступний приклад ілюструє обидва варіанти:

```
/* File ex4_16.c */
#include <stdio.h>
#include <glib.h>
```

```
int main() {
    GList *list = g_list_append(NULL, "Red ");
    list = g_list_append(list, "Blue ");
    list = g_list_append(list, "Blue ");
    list = g_list_append(list, "Green ");
    printf("Here's the list: ");
    g_list_foreach(list, (GFunc)printf, NULL);
    printf("\nItem 'Blue' is located at index %d\n",
         g_list_index(list, "Blue "));
    printf("Item 'White' is located at index %d\n",
         g_list_index(list, "White"));
    GList* last = g_list_last(list);
    printf("Item 'Green' is located at index %d\n",
         g_list_position(list, last));
    g_list_free(list);
}
```

Результат виконання:

```
Here's the list: Red Blue Blue Green
Item 'Blue' is located at index 1
Item 'White' is located at index -1
Item 'Green' is located at index 3
```

Зауважте, що виклик `g_list_index` повертає значення -1, якщо не може знайти дані. А якщо є два вузли з однаковими значеннями даних, `g_list_index` повертає індекс першого входження. Виклик `g_list_position` також повертає -1, якщо не може знайти вказаний вузол.

Знову ж таки, ці методи також присутні на `GSList` під різними назвами.

У цій главі ми ледь доторкнулися до можливостей, які надає бібліотека GLib. Увагу було приділено зв'язаним спискам (linked lists), а далі у цій бібліотеці на вас чекають бінарні дерева (binary trees), масиви, хеш-таблиці, кварки(quarks), списки даних із ключами (keyed data lists), n-арні дерева (n-ary trees) і ще багато чого цікавого і корисного, з чим вам слід буде самостійно розібратися, щоб додати нові потужні можливості до свого арсеналу розробника. Якщо ви готові відчути себе Алісою в Задзеркаллі, то переходьте за цим посиланням. Ця дорога приведе вас до товстунів Крутя і Вертя, проведе через сад квітів, які вміють розмовляти, і познайомить з Чорною Королевою...

Глава 5. Наукова бібліотека GSL

5.1. Огляд можливостей бібліотеки GSL

Назва бібліотеки “GSL” означає GNU Scientific Library. Про цю бібліотеку в оригінальній документації написано наступне. Наукова бібліотека GNU (GSL) – це набір програм для чисельних обчислень у прикладній математиці і наукових дослідженнях. Функції бібліотеки були написані з нуля на мові C і представляють собою сучасний інтерфейс прикладного програмування (API) для програмістів на мові C. Вихідний код розповсюджується за загальною публічною ліцензією GNU.

Перший реліз відбувся ще в далекому 1996 році, і весь цей час бібліотека невпинно розвивалася.

Бібліотека охоплює дуже широкий спектр тем з чисельних обчислень. Функції доступні для багатьох областей. Деякі з них приведу в якості приклада.

Комплексні числа (Complex Numbers), Корені поліномів (Roots of Polynomials), Вектори та матриці (Vectors and Matrices), Сортування (Sorting), Лінійна алгебра (Linear Algebra), Швидке перетворення Фур'є (Fast Fourier Transforms), Випадкові числа (Random Numbers), Гістограми (Histograms), Статистика (Statistics), Диференціальні рівняння (Differential Equations), Числове диференціювання (Numerical Differentiation), Інтерполяція (Interpolation), Пошук коренів (Root-Finding), Мінімізація (Minimization), IEEE з плаваючою точкою (IEEE Floating-Point), Фізичні константи (Physical Constants), Базисні сплайни (Basis Splines), Вейвлети (Wavelets) і ще багато чого цікавого і корисного для роботи справжнього інженера і науковця.

Правильне застосування цих можливостей ретельно описано в оригінальній документації, у кожній главі якої наведено детальні визначення функцій, а також приклади програм і посилання на наукові статті, на яких базуються реалізовані алгоритми.

5.2. Встановлення, компіляція та компонування

У попередній главі була розповідь про те, як встановити пакет у ОС Ubuntu. Є сподівання, що це вже знайома тема, тому без зайвих слів перейдемо до справи.

```
$ sudo apt update
$ sudo apt-get install libgsl-dev
```

```
$ dpkg -l | grep -i libgsl
ii  libgsl-dev         2.7.1+dfsg-3  amd64  GNU Scientific Library (GSL) – ➥
development package
ii  libgsl27:amd64     2.7.1+dfsg-3  amd64  GNU Scientific Library (GSL) – ➥
library package
ii  libgslcblas0:amd64 2.7.1+dfsg-3  amd64  GNU Scientific Library (GSL) – ➥
blas library package

$ find /usr -name 'libgsl*'
/usr/share/doc/libgslcblas0
/usr/share/doc/libgsl-dev
/usr/share/doc/libgsl27
/usr/lib/x86_64-linux-gnu/libgslcblas.so.0.0.0
/usr/lib/x86_64-linux-gnu/libgsl.a
/usr/lib/x86_64-linux-gnu/libgsl.so
/usr/lib/x86_64-linux-gnu/libgslcblas.so
/usr/lib/x86_64-linux-gnu/libgsl.so.27
/usr/lib/x86_64-linux-gnu/libgsl.so.27.0.0
/usr/lib/x86_64-linux-gnu/libgslcblas.so.0
/usr/lib/x86_64-linux-gnu/libgslcblas.a
```

Останні дві команди виконувати тепер необов'язково. Вони наведені лише в якості нагадування щодо попередніх наших зусиль. Якщо раптом ви щось не зрозуміли у наведеній послідовності команд shell або у вас щось не запрацювало як слід, то в цьому випадку слід звернутися до попередньої глави або безпосередньо до оригінальної документації за наступним посиланням https://www.gnu.org/software/gsl/doc/html/usage.html

```
$ cat example.c
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>
int main(void) {
 double x = 5.0;
 double y = gsl_sf_bessel_J0(x);
 printf("J0(%g) = %.18e\n", x, y);
 return 0;
}

$ gcc -Wall example.c -o example -lgsl -lgslcblas -lm
$ ls
example  example.c  example.o
```

```
$ ./example
J0(5) = -1.775967713143382642e-01
```

5.2. Генерація випадкових чисел

Все те, що стосується теми генерації випадкових чисел описано в оригінальній документації, яку ви можете знайти за посиланням https://www.gnu.org/software/gsl/doc/html/rng.html. Скористаємося цією документацією і напишемо програму, яка генерує кілька гаусових та гамма-випадкових чисел:

```
/* File gsl_random.c */
#include <stdio.h>
#include <gsl_rng.h>
#include <gsl_randist.h>

int main (int argc, char *argv[]) {
 /* set up GSL RNG */
 gsl_rng *r = gsl_rng_alloc(gsl_rng_mt19937);
 /* end of GSL setup */
 int n = atoi(argv[1]);
 for (int i=0; i<n; i++) {
     double gauss=gsl_ran_gaussian(r,2.0);
     double gamma=gsl_ran_gamma(r,2.0,3.0);
     printf("%2.4f %2.4f\n", gauss,gamma);
 }
 return(0);
}
```

```
$ gcc -Wall gsl_random.c -o gsl_random -lgsl -lgslcblas -lm

$ ls -l gsl*
-rwxrwxr-x 1 vadim vadim 16168 Jun 25 19:43 gsl_random
-rw-rw-r-- 1 vadim vadim   402 Jun 25 19:42 gsl_random.c

$ ./gsl_random 12
0.2678 6.9645
3.3488 1.6894
1.9950 2.1575
-4.7934 6.1648
-0.0782 4.0292
1.7871 11.6031
-2.5931 7.7629
```

```
0.3634 1.3344
-1.0965 11.1658
0.0142 3.5412
-1.6555 7.3112
-0.7352 6.5352
```

5.3. Прості векторні операції

У цій книзі чимало уваги мною було приділено саме матрицям і векторам. Бібліотеці GSL тут є що запропонувати навіть самому вибагливому розробнику. Залишилось продемонструвати ще декілька операцій, щоб надати вам змогу отримати перше враження:

```
/* File vectops.c */
#include <stdio.h>
#include <gsl/gsl_vector.h>

void printv1(gsl_vector *pv, int len);

int main(void) {
    const int LENGTH = 3;

    /* Create vector pvect1 */
    gsl_vector *pvect1 = gsl_vector_alloc(LENGTH);
    /* Create vector pvect2 */
    gsl_vector *pvect2 = gsl_vector_alloc(LENGTH);

    /* assign 3.2 to each element of vector pvect1 */
    gsl_vector_set_all(pvect1, 3.2);
    for (int i = 0; i < LENGTH; i++){
        printf("pvect1[%d] = %g\n", i, (pvect1->data)[i]);
    }

    double *ptrd = NULL;
    /* assign 12.5 to each element of vector pvect1 */
    gsl_vector_set_all(pvect2, 12.5);
    ptrd = pvect2->data;
    for (int i = 0; i < LENGTH; i++){
        printf("ptrd[%d] = %g\n", i, ptrd[i]);
    }

    printf("\nPrint data from vectors pvect1 and pvect2:\n");
```

```
for (int i = 0; i < LENGTH; i++) {
    printf("pvect1[%d] = %3.2f, pvect2[%d] = %3.2f\n", i,
    gsl_vector_get(pvect1, i), i, gsl_vector_get(pvect2, i));
}

double x = 3.14;
printf("\nAdd scalar %g to vector pvect1\n", x);
gsl_vector_add_constant(pvect1, x);
printv1(pvect1, LENGTH);

/* Compare two vectors */
if (gsl_vector_equal(pvect1, pvect2) == 1) {
    printf("Vectors pvect1 and pvect2 are equal\n");
} else {
    printf("Vectors pvect1 and pvect2 are not equal\n");
}

printf("\nAdd vector pvect2 to vector pvect1:\n");
gsl_vector_add(pvect1, pvect2);
printv1(pvect1, LENGTH);

printf("\nSubtraction of vector pvect2 from vector pvect1\n");
gsl_vector_sub(pvect1, pvect2);
printv1(pvect1, LENGTH);

printf("\nMultiply vector pvect1 by scalar %g\n", x);
gsl_vector_scale(pvect1, x);
printv1(pvect1, LENGTH);

printf("\nMultiplication of vectors pvect1 and pvect2\n");
gsl_vector_mul(pvect1, pvect2);
printv1(pvect1, LENGTH);

printf("\nDivision of vectors pvect1 and pvect2\n");
gsl_vector_div(pvect1, pvect2);
printv1(pvect1, LENGTH);

printf("\nCopy of vector pvect2 to pvect1\n");
gsl_vector_memcpy(pvect1, pvect2);
printv1(pvect1, LENGTH);

printf("\nSize of vector pvect1: %ld\n", pvect1->size);
x = 100.0;
printf("Assign %g to element 2 of pvect1\n", x);
```

```
    gsl_vector_set(pvect1, 2, x);
    printf("Max value in pvect1: %g\n", gsl_vector_max(pvect1));
    printf("Index of max: %ld\n", gsl_vector_max_index(pvect1));

    /* Release allocated memory */
    gsl_vector_free(pvect1);
    gsl_vector_free(pvect2);

    return 0;
}

void printv1(gsl_vector *pv, int len) {
    for (int i = 0; i < len; i++) {
        printf("pvect1[%d] = %3.2f\n", i, gsl_vector_get(pv, i));
    }
}
```

Для навчальних потреб було створено окремий Makefile:

```
# Simple Makefile for building C stuff with GSL
# CFLAGS=-Wall -I/usr/include/gsl
LDFLAGS=-lgsl -lgslcblas -lm
CC=gcc

%: %.c
	$(CC) $(CFLAGS) $< -o $@ $(LDFLAGS)

# e.g. do "make vectops" to make vectops from vectops.c

clean:
	rm -f *~ *.o core a.out
```

Спробуємо скористатися цим Makefile:

```
$ make vectops
gcc vectops.c -o vectops -lgsl -lgslcblas -lm

$ ./vectops
pvect1[0] = 3.2
pvect1[1] = 3.2
pvect1[2] = 3.2
ptrd[0] = 12.5
ptrd[1] = 12.5
ptrd[2] = 12.5
```

```
Print data from vectors pvect1 and pvect2:
pvect1[0] = 3.20, pvect2[0] = 12.50
pvect1[1] = 3.20, pvect2[1] = 12.50
pvect1[2] = 3.20, pvect2[2] = 12.50

Add scalar 3.14 to vector pvect1
pvect1[0] = 6.34
pvect1[1] = 6.34
pvect1[2] = 6.34
Vectors pvect1 and pvect2 are not equal

Add vector pvect2 to vector pvect1:
pvect1[0] = 18.84
pvect1[1] = 18.84
pvect1[2] = 18.84

Subtraction of vector pvect2 from vector pvect1
pvect1[0] = 6.34
pvect1[1] = 6.34
pvect1[2] = 6.34

Multiply vector pvect1 by scalar 3.14
pvect1[0] = 19.91
pvect1[1] = 19.91
pvect1[2] = 19.91

Multiplication of vectors pvect1 and pvect2
pvect1[0] = 248.84
pvect1[1] = 248.84
pvect1[2] = 248.84

Division of vectors pvect1 and pvect2
pvect1[0] = 19.91
pvect1[1] = 19.91
pvect1[2] = 19.91

Copy of vector pvect2 to pvect1
pvect1[0] = 12.50
pvect1[1] = 12.50
pvect1[2] = 12.50

Size of vector pvect1: 3
Assign 100 to element 2 of pvect1
```

```
Max value in pvect1: 100
Index of max: 2
```

А тепер вам слід завітати до сайту з офіційною документацією https://www.gnu.org/software/gsl/doc/html/vectors.html і продовжити розбиратися з іншими можливостями бібліотеки. Якщо для цього у вас все ще не з'явилося вагомої причини, то спробуйте пошукати в собі внутрішню мотивацію, яка дозволяє людині досліджувати нове без будь-якої конкретної причини, а просто з цікавості.

Маю вас попередити щодо коду, який використовує функції бібліотеки. На просторах інтернету ви можете знайти багато різних прикладів, але використовуючи ці приклади, не забувайте про те, що бібліотека живе і постійно вдосконалюється, тому деякі зразки коду можуть бути застарілими або взагалі некоректними. Щоразу, використовуючи ту чи іншу функціональність бібліотеки, не забувайте звірятися з офіційною документацією, наданою розробниками бібліотеки і уважно вивчайте повідомлення компілятора.

У 80-х роках минулого століття викладачі з інституту математичних досліджень університету Гренобля (місто на південному сході Франції) запропонували групі дітей вирішити наступну загадку: у човні 16 овець і 10 кіз; скільки років капітанові човна? Дивне питання. Який стосунок вік капітана має до кількості овець та кіз у його човні? Із приблизно 200 опитаних у віці від 7 до 8 років 75% дітей відповіли без жодних сумнівів. Багато хто просто склав представлені числа і отримав 26. Але, коли дітям віком від 9 до 10 років запропонували ту саму загадку, більшість із них почали протестувати або навіть відмовилися відповідати. Лише 20% відповіли беззастережно. За два роки їхнє критичне сприйняття загострилося. Ці діти стали проникливішими і почали сумніватися у сенсі того, що їм пропонують зробити. Дорослі програмісти інколи потрапляють в такі самі пастки, що й малі діти, але було б невірно вважати, що програмісти захищені від інших помилок, які чатують на них. Наших знать може бути недостатньо, наша інтуїція може нас обдурити, а дані виявитися хибними.

Післямова

Інколи можна почути такий внутрішньо суперечливий вислів: «Мові С вже багато років і чомусь на ній потрібно все писати з нуля». Люди, які висловлюють такі твердження, можливо не дали собі змоги зрозуміти, як робити компоновку програми з бібліотеками, про існування яких вони ніколи навіть і не чули. Скоріш за все, ви самі зустрічали подібні висловлювання. Але чи правдиві вони і якщо так, то якою мірою? Тепер ви самі можете у цьому розібратися і скласти власну думку з цього приводу. Адже одна і та сама людина може побачити щось як у негативному, так і в позитивному світлі, просто змінивши власну точку зору.

Зазвичай подібні книги закінчуються словами: «Я сподіваюся, що моя книга вам сподобалася і ви чогось навчилися», але такі слова є зайвими. Якщо ви справді щось зрозуміли і дійсно чогось навчилися, розібравшись у матеріалі цієї книги, то ви самі це відчуєте або не відчуєте, а бідь-які сподівання – це лише чиїсь сподівання і вони тут зовсім ні до чого.

Як відчути силу своїх знань, які ви щойно отримали? Це доволі просто. Риба не підозрює про те, що вона знаходиться у воді, доки її звідти не витягнеш. Ви маєте вихідний код перших трьох глав цієї книги, який було написано для вас, спираючись виключно на можливості стандартної бібліотеки мови С. В свою чергу вам був показаний шлях і запропоновано розібратися з різноманітними можливостями двох популярних бібліотек. Ось вам достойний виклик: реалізуйте алгоритм решітки Кардано та нейронну мережу, що розпізнає рукописні цифри, за допомогою функцій, що надаються у ваше розпорядження двома бібліотеками GLib і GSL.

Все в житті, чого варто прагнути, здобувається через негативний досвід. Будь-які спроби уникнути чи заглушити негатив лише дадуть зворотний ефект. Спроба уникнути страждання сама по собі вже є формою страждання. Намагання уникнути боротьби є боротьба. Відмова визнати невдачу є невдачею. Приховування ганебного саме є ганебним. Все своє життя треба старанно вчитися, щодня ставати більш майстерними, ніж ви були за день до цього, а наступного дня – більш майстерними, ніж сьогодні. Вдосконаленню немає кінця.

Бажаю успіхів!

Рекомендована література

1. Neural Networks for Beginners: Unlock the Secrets of Neural Networks. A Beginner's Guide to AI's Most Powerful Tool by James Ferry, 2024. URL: Neural Networks for Beginners
2. C Programming Language, 2nd Edition by Brian W. Kernighan, Dennis M. Ritchie. URL: C Programming Language, 2nd Edition
3. C: A Reference Manual, 5th Edition 5th Edition by Guy Steele Jr., Samuel Harbison. URL: C: A Reference Manual, 5th Edition 5th Edition
4. C Pocket Reference: C Syntax and Fundamentals 1st Edition by Peter Prinz (Author), Ulla Kirch-Prinz. URL: C Pocket Reference: C Syntax and Fundamentals 1st Edition
5. Standard C Library, The First Edition by P.J. Plauge. URL: Standard C Library, The First Edition
6. Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching 3rd Edition. URL: Algorithms in C, Parts 1-4
7. C Programming by Manmohan Singh, Rahul Sharma, Neeraj Kumar Rathore, Urmila S Soni.Arcler Press, 2024. URL: C Programming

Перелік посилань

1. Річард Фейнман. URL: https://en.wikipedia.org/wiki/Richard_Feynman
2. Метод шифрування за допомогою решітки Кардано. URL: https://en.wikipedia.org/wiki/Grille_%28cryptography%29
3. Метод «грубої сили». URL: https://en.wikipedia.org/wiki/Brute-force_search
4. Посилання на вихідний код до глави 1. URL: https://github.com/iamvadimov/grille
5. Ян ЛеКун. URL: http://yann.lecun.com/
6. База даних рукописних цифр MNIST. URL: https://yann.lecun.com/exdb/mnist/
7. Тренувальний набір для нейронної мережі. URL: https://www.pjreddie.com/media/files/mnist_train.csv
8. Тестовий набір для нейронної мережі. URL: https://www.pjreddie.com/media/files/mnist_test.csv
9. Книга «Глибоке навчання». URL: https://www.deeplearningbook.org/
10. Посилання на вихідний код до глави 3. URL: https://github.com/iamvadimov/cmnist
11. Бібліотека Glib. URL: https://docs.gtk.org/glib/
12. Опис POSIX. URL: https://uk.wikipedia.org/wiki/POSIX
13. Більше інформації про Make. URL: https://www.gnu.org/software/make/
14. Більше відомостей по Glib. URL: https://docs.gtk.org/glib/index.html
15. Більше відомостей по shell командам. URL: https://www.gnu.org/software/gsl/doc/html/usage.html
16. Генерація випадкових чисел. URL: https://www.gnu.org/software/gsl/doc/html/rng.html
17. Офіційна документація: вектори та матриці. URL: https://www.gnu.org/software/gsl/doc/html/vectors.html