Типи даних, що визначені користувачем. Структури. Робота з файлами

Структури

Структура — це сукупність різнотипних елементів або логічно зв'язаних змінних, в який входять елементи будь-яких типів (на старому Сі - за винятком функцій, хоча можна використовувати вказівник на функції, на сучасному Сі та Сі++ можна використовувати й функції), яким присвоюється одне ім’я (на мові Сі воно може бути відсутнім – так звана *анонімна структура*) для зручності подальшої обробки , що займає одну ділянку пам’яті, тобто визначено як окремий тип. Елементи, що складають структуру, називаються ***полями***.

На відміну від масиву, який є однорідним об'єктом, структура може бути неоднорідною.

Традиційним прикладом структури служить облікова картка того, що працює: службовець підприємства описується набором атрибутів, таких, як табельний номер, ім'я, дата народження, стать, адрес, зарплата. У свою чергу, деякі з цих атрибутів самі можуть виявитися структурами. Такі, наприклад: ім'я, дата народження, адрес, що мають декілька компонент.

Декларація структури

Змінна типу структура, як і будь-яка змінна, повинна бути описана. Цей опис складається з двох кроків: опису шаблону (тобто складу) або типу структури та опису змінних структурного типу.

Синтаксис декларації структури:

struct [<tag>] {
<member-declaration-list>}
<declarator>
[, <declarator> ...];

або

struct <tag> <declarator> [, <declarator> ...];

Декларація структури задає ім'я типу структури і специфікує послідовність змінних величин, що називаються полями (елементами, членами) структури. Ці поля структури можуть мати різні типи.

Декларація структури починається з ключового слова struct і має дві форми подання, як показано вище. У першій формі подання типи і імена елементів структури специфікуються в списку декларацій елементів <member-declaration-list>. <tag> - це ідентифікатор, який іменує тип структури, визначений у списку декларацій елементів.

Кожен <declarator> задає ім'я змінної типу структури. Тип змінної в декларатор може бути модифікований на покажчик до структури, на масив структур або на функцію, яка повертає структуру.

Друга синтаксична форма використовує тег <tag> структури для посилання на тип структури. У цій формі декларації відсутній список декларацій елементів, оскільки тип структури визначений в іншому місці. Визначення типу структури

має бути видимим для тегу, який використовується в декларації і визначення повинне передувати декларації через тег, якщо тег не використовується для декларації вказівника або структурного типу typedef. В останніх випадках декларації можуть використовувати тег структури без попереднього визначення типу структури, але все ж визначення повинне знаходитися в межах видимості декларації.
Список декларацій елементів <member-declaration-list> - це одне або більше декларацій змінних або бітових полів. Кожна змінна, що об'явлена в цьому списку, називається елементом структурного типу. Декларації змінних списку мають той же самий синтаксис, що і декларації змінних за винятком того, що вони не можуть містити специфікаторів класу пам'яті або ініціалізаторів.
Елементи структури можуть бути будь-якого типу: будь-якого базового, масивом, вказівником, об'єднанням або структурою.
Єдине обмеження - поле не може мати тип батьківської структури, в який воно описано. Однак, поле може бути описано, як вказівник на тип структури, до якої він входить, дозволяючи створювати рекурсивні структури, наприклад списки.
Таким чином, опис структури має вигляд:

```
struct [<ім'я структури>]
{ <тип 1> ім'я поля 1;
  <тип 2> ім'я поля 2 . . .;
} p1, p2 . . .;
```

де struct — службове слово;

- **<ім'я структури>** — ім'я типу структура (може бути відсутнім);
- **<тип 1>, <тип 2>** — імена стандартних або визначених типів;
- **ім'я поля 1, ім'я поля 2,…** — імена полів структури;
- **p1, p2 . . .;** — імена змінних типу структура.

Таким чином, створюється змінна нового типу struct [<ім'я структури>] який можна використовувати в програмі як новий тип, передавати як аргумент та повертати з функції. *Однак, зауважимо, що оскільки це є новим типом для нього за замовченням не визначено жодної стандартної функції окрім присвоювання і таким чином потрібно для нової структури визначати потрібні операції додатково*.
Примітка: На Сі++ можна користуватись структурою без використання ключового слова struct [<ім'я структури>] , а лише вказавши <ім'я структури> у якості типу нової змінної у декларації.
Наприклад, для зберігання інформації по успішності студента з дисципліни «Програмування» визначимо таку структуру:

```
struct Student  // визначаємо структуру з назвою Student

{ char  name [25]; //  Поле 1:  Прізвище та ініціали – тип рядок з 25 символів
  char group[3]; //  Поле 2:  Група – тип рядок з 3 символів
int pract_mark; //  Поле 3:  Бали за практику – тип ціле
```

```
int course_project1; //  Поле 4:  Бали за перший проект – тип ціле
int course_project2; //  Поле 5:  Бали за другий проект – тип ціле
float additional_mark; // Поле 6: середній додатковий бал  -тип дійсне число
}  st1, st2;
```

Ця декларація може бути зроблено як локально так і глобально, і це означає що ми створили дві змінні типу struct Student.

Змінні st1 і st2 можна оголосити окремим оператором, наприклад:

```
// створюємо глобально структуру з назвою Student
struct Student  {
char  name [25]; //  Поле 1:  Прізвище та ініціали – тип рядок з 25 символів
char group[3]; //  Поле 2:  Група – тип рядок з 3 символів
int pract_mark; //  Поле 3:  Бали за практику – тип ціле
int course_project1; //  Поле 4:  Бали за перший проект – тип ціле
int course_project2; //  Поле 5:  Бали за другий проект – тип ціле
float additional_mark; // Поле 6: середній додатковий бал  -тип дійсне число
} ;
//Далі в коді глобально або локально визначаємо відповідні змінні
struct Student stl, st2;.
```

Приведена в прикладі структура має три частини, які називаються елементами або полями. Для того, щоб працювати зі структура необхідно засвоїти три основні прийоми:

- встановлення формату структури;
- визначення змінної, яка відповідає даному формату;
- забезпечення доступу до окремих компонентів змінної-структури.

Розглянемо наступний приклад.

```
struct book {
char nazva[maxnazva];
char avtor[maxavtor];
float cina;
};
```

Опис структури, що складається з взятого в фігурні дужки списку описів, починається з службового слова struct. За словом struct може записуватися необов’язкове ім’я, яке називається назвою структури (тут це book). Ярлик іменує структури і може використовуватись надалі як скорочений запис докладного опису. Список полів(елементів) знаходиться в фігурних дужках. Кожне поле має свою назву. Після визначення кожного елемента ставиться крапка з комою. Поле структури має будь-який тип даних, а також може включати в себе інші структури. Опис структури завершується крапкою з комою. Опис структури може бути розташований ззовні функції і всередині. Якщо опис поміщено всередину функції , то структура використовується лише всередині функції. Поняття структура ” може використовуватись в двох значеннях. Одно з них-шаблон. Шаблон вказує компілятору, як представити дані, але для них не виділяється пам’ять, він лише визначає форму структури.

Імена елементів і тегів без яких-небудь колізій можуть співпадати з іменами звичайних змінних (тобто не елементів), оскільки вони завжди помітні по контексту. Більш того, одні і ті ж імена елементів можуть зустрічатися в різних структурах, хоча, згідно хорошого стилю програмування, краще однакові імена давати тільки близьким по сенсу об'єктам.
Приклад:
Так, наприклад для анкети службовця можна вибрати такі імена:
tab_nom - табельний номер;
pib - прізвище, ім’я, по батькові;
stat- стать;
summa - зарплата;
Всі ці поняття можна об'єднати в таку, наприклад, структуру:

```
struct anketa {
int tab_nom;
char fio[30];
char data[10];
int pol;
char adres[40];
float summa;};
```

Друге значення, поняття “структура” - це змінна - структура, яка створюється на наступному етапі. Створення структурної змінної робиться за допомогою такого опису:

```
struct book odna;
```

Обробляючи цей оператор, компілятор створює змінну odna і використовуючи шаблон book виділяє пам’ять для двох символьних масивів змінної дійсного типу. При визначенні змінної-структури шаблон book відіграє таку ж роль, як слова int і float для більш простих описів. Для комп’ютера визначення

```
struct book odna;
```

є скороченим варіантом опису:

```
struct book {
char nazva[maxnazva];
char avtor[maxavtor];
float cina;
} odna;
```

Можна визначити декілька змінних –структур або вказівник на цей вигляд структури:

```
struct book odna, libry, *pbook;
```

Ще один класичний приклад:

```
1) // Декларація структури Точка.
struct Point
{
   int x, y;  // однотипні поля можна обєднувати як змінні через кому
} p1;  // Змінна p1 декларована як 'Point'
```

2)
```
// Створення нового типу Точка як базового типу
struct Point {
        int x, y;
};
int main() {
  struct Point p1;  // Змінна p1 декларована як звичайна змінна
}
```
Примітка: В **C++** ключове слово struct *необов'язково* при декларації змінної. В C воно ***обов'язкове***.

Ініціалізація

Розглянемо тепер ініціалізацію структур і структурних змінних. До останніх версій Сі++ не можна було ініціалізувати структуру безпосередньо під час декларації. Наприклад, наступна C програма на Сі -компіляторах повинна впасти.

```
struct Point
{
        int x = 0;  // COMPILER ERROR:  cannot initialize members here
        int y = 0;  // COMPILER ERROR:  cannot initialize members here
};
```

Причина подібної поведінки наступна — коли тип об'явлений, пам'яті під нього ще не виділено. Пам'ять виділяється лише коли створюються відповідні змінні.
В нашому прикладі визначення структури є зовнішнім, а змінна структурного типу описана всередині функції . Для ініціалізації структури використовується синтаксис за допомогою фігурних дужок, подібний тому, який використовується при ініціалізації масивів:

```
struct book odna={“Три мушкетери”, А.Дюма, 3};
```

або

```
struct Point {
   int x, y;
};

int main() {
        // Ініціалізація. Полю x присвоюється значення  0,  а  y  присвоюється 1.
        //  Порядок декларації визначає порядок ініціалізації.
        struct Point p1 = {0, 1};
}
```

Отже, використовується список ініціалізаторів розділених комами і взятий у фігурні дужки. Кожне конкретне значення (ініціалізатор) повинен відповідати типу елемента структури, якому присвоюється початкове значення. Тому

можна одночасно присвоїти елементу nazva стрічкове значення, а елементу cina-числове. Для ясності кожному елементу відводиться власна стрічка ініціалізації, але компілятору достатньо того, щоб значення були розділені комами.
Ініціювання полів структури слід здійснювати або при її описі, або в тілі програми. При описі структури ініціювання полів виглядає, наприклад, для описаної вище структури Student
st1 = {"Молодець І. І.", 40, 10, 10, 3.4f};
st2 = {"Поганенко А. М.", 20, 1, 1, -3.5f};
Якщо ініціювання виконується в тілі програми, то для звернення до імені поля треба спочатку записати ім'я структурної змінної, а потім ім'я поля. Ці обидва записи відокремлюються крапкою і являють собою складене ім'я.
Отже, у випадку появи змінної st1 у програмі для її ініціювання можна записати ініціалізацію за допомогою фігурних дужок, або ініціювання виконується за допомогою доступу до кожного поля.

```
int main() {
        struct Point p1;
        // Доступ до полів point p1
        p1.x = 20;
        p1.y = 10;
        printf ("x = %d, y = %d", p1.x, p1.y);
}
```

Направлена Ініціалізація (Designated Initialization) дозволяє ініціалізувати поля структури в будь-якому порядку. Ця властивість додана в Сі (не Сі++) зі стандарту C99 .

```
#include<stdio.h>

struct Point3D {
 int x, y, z;
};

int main()
{
         // приклад designated initialization
        struct Point3D p1 = {.y = 0, .z = 1, .x = 2};
        struct Point3D p2 = {.x = 20};

        printf ("x = %d, y = %d, z = %d\n", p1.x, p1.y, p1.z);
        printf ("x = %d", p2.x);
        return 0;
}
```

Результат роботи:
x = 2, y = 0, z = 1
x = 20
ця властивість відсутня в С++ тобто присутня лише в С.
Серед полів можна використовувати й вказівники.

Розглянемо ілюстраційну програму:

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

int main ( ){
        // визначили структуру
        struct credit { char* pib; int theory[2]; int tasks[2]; float avg;} st1, st2;
        // потрібно виділити пам'ять під перше поле
        st1.pib=(char*) malloc(30);
        strcpy (st1.pib, "Нездавайло Х.Х.");// та ініціалізувати його
        st1.theory = {2,3}; // ініціалізуємо інші поля
        st1.tasks[0] = 0;
        st1.tasks[1] =3;
// рахуємо середній бал
st1.avg = (float) (st1.theory[0] +st1.theory[1] + st1.tasks[0] + st1.tasks[1]);
        st2 = st1;  // присвоєння структур - визначено
        puts (st2.pib); // виводимо 1 поле 2-ої структури
free(st1.pib); /* звільнюємо ділянку під прізвище – це звільнить й друге
прізвище!!*/

// 2 варіант – більш правильний
        st1.pib=(char*) malloc(30); // виділямо дві ділянки під призвища
        st2.pib=(char*) malloc(30);
        strcpy (st1.pib, "Здавайло У.У");
        strcpy(st2.pib,st1.pib, sizeof(st1.pib));

        free(st1.pib); // звільнюємо дві ділянки під призвища
        free(st2.pib);
}
```

У наведеній програмі організується присвоювання всім полям структури st1 відповідних значень. Слід зауважити, що поле st1.pib одержує значення шляхом застосування функції strcpy (). Структурна змінна st2 того ж типу, що і st1, тому справедлива операція st2 = st1;.

Анонімна структура

Якщо функція використовує тільки один структурний тип, то цей тип можна оголосити без імені. Тоді раніше розглянуту структуру можна оголосити таким чином:

```
struct{ char fam [25];

         int mat, fiz, prg;
        float sb;
} stl, st2;
```

Коли при описі структур у деякій функції або в межах видимості змінних у різних функціях є багато (але не всі) однакових полів, то їх слід об'єднати в окрему структуру. Її можна застосовувати при описі інших структур, тобто поля структури можуть самі бути типу struct. Це називається вкладеністю структур — її можна використати, наприклад, якщо треба обробляти списки студентів та викладачів університету. Студентські списки містять дані: прізвище та ініціали, дата (день, місяць, рік) народження, група та середній бал успішності, а в списках викладачів присутні такі дані: прізвище, ініціали, дата народження, кафедра, посада. У процесі обробки списку студентів і списку викладачів можна оголосити відповідно такі структури:

```
struct stud{ char fio [25];
 int den, god;
 char mes [10];
 char grup;
 float sb; }
struct prep
{ char fio [25];
int den, god;
char mes [10];
char kaf, dolg;
}
```

В оголошених типах однакові поля можна об'єднати в окрему структуру і застосовувати її при описі інших типів. Поетапно це виглядає так:

- загальна структура:

```
struct spd
{ char fio [25];
int den, god;
char mas[10]; }
```

- структура для опису інформації про студентів :

```
struct stud
{ spd dr;
char grup;
float sb}
st1, st2;
```

- структура для опису інформації про викладачів:

```
struct prep
{ spd dr;
char kaf [10];
char dolg [15];
} pr1, pr2;
```

У структурах stud і prep для оголошення поля, що містить дані про прізвище і дату народження, використовується раніше описаний тип spd. Тепер до поля

fio, den, god, mes можна звернутися, використовуючи запис st1.dr.fio, наприклад, при зверненні до функції введення:
gets (st1.dr.fio);
або gets (pr1.dr.fio);.

Визначення власного типу

В мові С є ключове слово **typedef**, яке можна використовувати для визначення власного типу даних. Наприклад таким чином можна зробити щось на зразок аліасингу Пітона, наприклад визначити типе **BYTE** для однобайтового цілого:
typedef unsigned char BYTE;
Після цього визначення BYTE може використовуватись для визначення типу unsigned char**, наприклад:**
BYTE b1, b2;
За домовленістю (стилем коду), заголовочні літери (uppercase letters) використовуються в таких означеннях щоб вказувати користувачам програми що цей тип насправді абревіатура, але насправді можна використовувати й звичайні літери
typedef unsigned char byte;
Можна і в багатьох випадках потрібно використовувати **typedef** для того щоб декларувати новий тип даних, що ми власне створили в програмі, зокрема при визначенні структури. Наприклад:

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

typedef struct Books {
  char title[50];
  char author[50];
  char subject[100];
  int book_id;
} Book;

int main( ) {
  Book book;
  strcpy( book.title, "C Programming");
  strcpy( book.author, "Cool Teacher");
  strcpy( book.subject, "Programming");
  book.book_id = 111111;

  printf( "Book title : %s\n", book.title);
  printf( "Book author : %s\n", book.author);
  printf( "Book subject : %s\n", book.subject);
  printf( "Book book_id : %d\n", book.book_id);
```

```
  return 0;
}
```
Результат роботи:
```
Book  title : C Programming
Book  author : Cool Teacher
Book  subject : Programming
Book  book_id : 111111
```

Вказівники на структури

Існує, принаймні, три причини використання вказівників на структури:

- вказівниками легше керувати;
- структура не може передаватись функції в якості аргументу, але це можливо для вказівника;
- в деяких оголошеннях даних використовуються структури, які містять вказівники на інші структури.

Оголошення вказівника на структуру:

struct book *dn;

Спочатку йде службове слово struct ,потім назва структури book , зірочка і ім'я вказівника. Це оголошення не створює нову структуру, але дозволяє використовувати вказівник dn для позначення будь-якої існуючої структури типу book.

Вказівник ініціалізується таким чином, що вказує на структуру.

dn=&x1;

Розглянемо приклад:

```
#include <stdio.h>
#define N 20
typedef struct names{ /*перший шаблон*/
char imya[N];
char prizv[N];
} names;
typedef struct harakter { /*другий шаблон*/
struct names druzi; /*вкладена структура*/
char bludo[N];
char robota[N];
float zarob;
} harakter;
int main() {
struct harakter x1[2]{{ /*ініціалізація змінної*/
{“Іван” ,“Петренко”}, “вареники”, “інженер”,30250.00},
{“Петро”,”Іващенко”},”борщ”,”лікар”,40325.00}};
```

```
struct harakter *x2;
x2=&x1[0]; /*вказує на структуру*/
printf(“заробіток: %lf \n”,x2->zarob);
x2++; /*вказує на наступну структуру*/
printf(“заробіток: %lf \n”, (*x2).zarob);
printf(“улюблена страва %s - %s  \n ”,x2→druzi.prizv,x2->bludo);
}
```

Вказівник x2 вказує на елемент x1[0]. Можна використати цей вказівник для отримання значення елемента x1[0].

1. За допомогою нової операції ->. Ця операція складається з дефіса (-) і символу “більше”(>). Іншими словами, вказівник структури з операцією -> має такий самий зміст, які ім’я структури, після якого застосовується операція “крапка”.
2. Якщо **x2= =&x1[0],**то і ***x2= =x1[0],** оскільки **&** і * є еквівалентними операціями. Тому можлива заміна

x1[0].zarob= =(*x2).zarob

Дужки необхідні, оскільки пріоритет операції “крапка” вище, ніж пріоритет операції “зірочка”.

Так само, як і для базових типів, після оголошення структури як певного типу можна створювати масиви структур.

```
#include<stdio.h>
struct Point
{
        int x, y;
};
int main() {
    // Масив структур
     struct Point arr[10];
      // Доступ до членів масиву
        arr[0].x = 10;
        arr[0].y = 20;

        printf("%d %d", arr[0].x, arr[0].y);
        return 0;
}
```

Результат роботи:

10 20

Після оголошення структурного типу змінних для роботи з їхніми полями можна застосовувати і вказівники, тоді опис структури матиме вигляд:

```
struct stud
{ char fam [25];
  int mat, fiz, prg;
  float sb;
} st1, *pst;
```

Доступ до полів може здійснюватися двома способами:

- з використанням операції розіменування «*», тобто
  gets ((*pst).fam); (*pst).fiz = 5;
- з використанням вказівника ->, наприклад,
  gets (pst -> fam); pst-> fiz = 5; тощо.

Крім того, до полів змінної st1 можна звертатися, вказуючи поля через операцію «**.**», як це робилося раніше.

Дані типу структура можна об'єднати в масиви, тоді для розглянутого вище ілюстраційного прикладу з урахуванням кількості студентів, структуру можна записати так:

```
struct stud
{ char fam [20];
        int mat, fiz, prg;
        float sb;
} spis[15], *sp = &spis[0];.
```

У випадку, коли масив описується десь у тексті програми, тобто не саме після опису структури, його можна оголосити у вигляді: stud spis [15]; — масив типу структура з ім'ям stud, що містить відповідну інформацію про групу із 15 студентів.

Доступ до елементів масиву типу структура здійснюється із застосуванням індексу або через покажчик-константу, яким є ім'я масиву, тобто одним з таких способів:

```
strcpy (spis[1].fam, " ");
spis[1].fiz = 5;
```

або

```
strcpy ((sp + 1) -> fam, " ");
```

Приведена в прикладі структура має три частини, які називаються елементамиабо полями. Для того, щоб працювати зі структура необхідно засвоїти три основні прийоми:

- *встановлення формату структури;*
- *визначення змінної, яка відповідає даному формату;*
- *забезпечення доступу до окремих компонентів змінної-структури*.

Поля структури можуть також бути масивами. Наприклад, у розглянутій структурі stud можна оцінки з різних предметів об'єднати в масив. Тоді структуру слід описати у вигляді:

```
struct stud1
{ char fam [25];
```

int pred [3];
float sb
} spis[15], *ps = &spis[0];.
Звернення до полів здійснюватиметься одним із способів:
((*ps).fam)
(ps->pred [0]),
наприклад,
gets ((*ps).fam);
printf(“%d %d %d”, ps -> pred[0] , ps -> pred[1] , ps -> pred[2]);
або
printf(“%d %d %d”,ps -> *(pred + 0) >> ps -> *(pred + 1) >> ps -> *(pred + 2)).
Доцільно також зазначити, що *бібліотека stdlib.h містить спеціальні функції для пошуку та сортування структурних змінних.*

Ще приклад:

```
#include<stdio.h>
typedef struct Point {
        int x, y;
}Point;
int main() {
        Point p1 = {1, 2};  // визначили змінну типу Point
        Point *p2 = &p1;  // p2  -  це вказівник на структуру p1
        // Доступ до полів структури використовуючи вказівник
        printf("%d %d", p2->x, p2->y);

        Point * p_array; //  визначили безрозмірний масив типу Point
        unsigned n;
        scanf(«%u»,&n); // ввели кількість елементів масиву p_array
        p_array = ( Point *) malloc(n*sizeof(Point)); // виділили память

        // ввели масив точок
        for(int i=0;i<n;i++){
                scanf(« %d %d », p_array[i].x, p_array[i].y);
        }
        int i=0;
        Point * ptr = p_array;
        while (i<n){ // вивели масив точок ітеруючись по вказівнику
                 i++;
                 printf(«%d %d», ptr→x, ptr->y);
                 ptr++;
        }
        free(p_array); // не забули звільнити память
}
```

Робота з файлами

Основні концепції роботи з файлами

Файл *– це поіменована сукупність даних, яка зберігається на пристрої.* Будь-який файл являє собою послідовність байтів. В залежності від того, який сенс та призначення мають ці байти, файли поділяють на текстові та бінарні (двійкові). Таким чином, файли поділяються на текстові та бінарні.
Текстовий файл - текстовий потік даних, фактично рядок, що має ім'я та зберігається на пристрої. Текстові файли містять байти, що є кодами алфавітних, цифрових символів та знаків пунктуації (пробілів, табуляцій та символи переходу на новий рядок)
Бінарний файл – це сукупність будь якого типу даних, що так само знаходиться в пристрої. Бінарні файли можуть містити будь-які значення байтів, в тому числі і такі, яким не відповідає графічний знак, що може бути виведений на екран. При цьому для того, щоб зчитувати конкретний файл, потрібно знати які дані, в якому порядку записані на пристрої.
Потоки вводу-виводу - це об'єкти типу FILE, до яких можна отримати доступ і маніпулювати ними лише за допомогою вказівників типу FILE *
Примітка: Хоча можна створити локальний об'єкт типу FILE, використання адресу такої копії змінної вводу-виводу в функції є невизначеною поведінкою, тобто не визначено стандартом і може призвести до серйозних помилок.
Кожен потік вводу виводу пов'язаний з зовнішнім фізичним пристроєм (*файл, стандартний вхідний потік, принтер, послідовний порт* тощо).
Потоки вводу-виводу можуть використовуватися як для неформатованих, так і для форматованих входів і виходів. Вони чутливі до локалі і можуть виконувати широкі / багатобайтові перетворення, якщо це необхідно. Всі потоки отримують доступ до одного і того ж локального об'єкта: останнього встановлено з setlocale.
Окрім специфічної для системи інформації, необхідної для доступу до пристрою (наприклад, дескриптор файлу POSIX), кожен об'єкт потоку містить наступне:
1) **Ширину символів (починаючі зі стандарту** C95**)**: *не встановлену, вузьку* або *широку*(unset, narrow або wide)
2) **Стан буферизації**: *небуферизований, лінійний буфер, повністю буферизований*.(unbuffered, line-buffered або fully buffered)
3) **Буфер**, який може бути замінений зовнішнім, наданим користувачем буфером.
4) **Режим введення / виводу:** *введення, виведення* або *оновлення* (вхід і вихід).
5) **Індикатор бінарного / текстового режиму**.
6) **Індикатор стану кінця файлу**.
7) **Індикатор стану помилки.**
8) **Індикатор положення файлу** (об'єкт типу fpos_t), який для широких потоків символів включає стан розбору (об'єкт типу mbstate_t (C95)).

9) Блокування повторного вмикання, що використовується для запобігання гонкам даних, коли кілька потоків читають, записують, розміщують або запитують позицію потоку (починаючі зі стандарту C11).

Вузька і широка орієнтація
Відкритий потік не має орієнтації. Перший виклик у fwide або до будь-якої функції вводу-виводу встановлює орієнтацію: широка функція введення-виведення робить потік з широким орієнтуванням, вузька функція вводу-виводу робить потік вузьким. Після встановлення ширини її можна змінити лише за допомогою freopen. Звичайні (вузькі) функції вводу-виводу не можна викликати на широкому потоці. Широкі функції вводу-виводу не можна викликати на вузько орієнтованому потоці. Широкі функції вводу-виводу перетворюють між широкими і багатобайтовими символами за допомогою виклику mbrtowc і wcrtomb. На відміну від багатобайтових символьних рядків, які є дійсними в програмі, багатобайтові послідовності символів у файлі можуть містити вбудовані нулі і не повинні починатися або закінчуватися в початковому стані зсуву. POSIX вимагає, щоб параметр LC_CTYPE поточно встановленої локалі C зберігався в потоковому об'єкті в той момент, коли його орієнтація стала широкою, і використовується для всіх майбутніх входів / виходів цього потоку, поки не буде змінена ширина, незалежно від будь-яких наступних викликів до setlocale .

Текстовий потік є впорядкованою послідовністю символів, що складається з рядків (нуль або більше символів плюс закінчується '\n'). Чи вимагає останній рядок закінчення '\n', визначено реалізацію. Символи, можливо, повинні бути додані, змінені або видалені на вході і виході, щоб відповідати умовам для представлення тексту в ОС (зокрема, потоки C на ОС Windows перетворюються на \n на виході, і конвертуються \r\n на вході). Дані, прочитані з текстового потоку, гарантовано порівнюються з даними, які раніше були виписані в цей потік, лише якщо виконується наступне:

- дані складаються тільки з друкованих символів і контрольних символів і спецсимволів n (зокрема, в ОС Windows, символ \t та \n відразу передує символ пробілу (пробіли, які виписуються безпосередньо перед останнім символом \0x1A;
- ніякого \n немає безпосередньо перед символом пробілу (пробіли, які виписуються безпосередньо перед \n знищуються
- символ кінця рядку - \n

Бінарний (двійковий) потік є впорядкованою послідовністю символів, яка може прозоро записувати внутрішні дані. Дані, що зчитуються з двійкового потоку, завжди дорівнюють даним, які були раніше записані в цей потік. Реалізаціям дозволено лише додавати до кінця потоку ряд нульових символів. Широкий двійковий потік не повинен закінчуватися в початковому стані зсуву. У реалізаціях POSIX не розрізняють текстові та двійкові потоки (спеціального відображення для будь-яких інших символів немає)

З точки зору програміста робота з файлами виглядає так:
1) Програма оперує не з маркером магнітного диску для зчитування інформації, а з допоміжною змінною (скажемо, f) типу вказівник на певну

структуру, через які вже операційна система отримує доступ до конкретного місця на диску.

2) Перш ніж робити з файлом на диску будь-які дії (читати дані з файлу чи писати файл), програма повинна його відкрити. Файл, розташований на диску, зв'язується зі змінною f (тобто до змінної заносяться службові дані для доступу саме до даного файлу).

3) Після цього, коли треба записати чи прочитати дані з файлу, програміст викликає спеціальні функції читання та запису, передаючи їм через один аргумент змінну f канал доступу до файлу, а через решту аргрментів – які саме дані читати чи писати.

4) На закінчення роботи програма повинна закрити файл і розірвати зв'язок між змінною f та файлом на диску, при цьому звільняються ресурси операційної системи, що використовруються для доступу до файлу.

Основні функції файлового введення-виведення оголошено в тому ж заголовочному файлі stdio.h, що й функції введення з клавіатури та виведення на екран. Вся подальша робота з файлом здійснюється через змінну f та вказівник на структуру даних типу FILE.

Відкриття файлу

Значення, яке повертає функція fopen – вказівник на службову структуру даних, через яку програма може звертатися до файлу на диску, присвоюється змінній f. Відтепер ця змінна є посередником між програмою та дисковим пристроєм.

Відкриття файла здійснюється за допомогою функції **fopen**, яка має наступний формат:

FILE* fopen(const char* filename, const char* mode);

тобто

FILE *fopen (“фізичне ім’я файлу”, “режим”);

Тут **FILE** - ім’я типу, описане в заготовочному файлі **stdio.h**;– вказівник на цю структуру.

Функція **fopen** повертає вказівник на файлову змінну, або **NULL**, якщо під час відкриття файла виникла помилка. Фізичне ім’я файлу – це його повне ім’я, включаючи шлях до файлу.

Можливі режими:

“**w”**- текстовий файл відкривається для запису; якщо файл із вказаним іменем існує, то його вміст знищується;

“**r”**- текстовий файл відкривається для читання;

“**a”**- текстовий файл відкривається для доповнення, інформація додається в кінець файлу.

Таблиця 5.1

Параметри доступу до файлу

Тип	Опис
r	Читання. Файл повинен існувати.
w	Запис нового файлу. Якщо файл с з таким іменем вже існує, то його зміст

буде знищено. Інакше він створиться

a	Запис в кінець файлу. Операції позіціонування (fseek, fsetpos, frewind) ігноруються. Файл створюється, якщо не існував.
r+	Читання та оновлення. Можно як читати, так і записувати. Файл повинен існувати.
w+	Запис і оновлення. Створюється новий файл. Якщо файл с з таким іменем вже існує, то його зміст буде знищено. Можно як читати, так і записувати.
a+	Запис в кінець файлу й оновлення. Операції позіціонування працюють лише для читання, для запису ігноруються. Якщо файлу не існувало, то він створиться.

Якщо необхідно відкрити файл в бінарному режимі, то в кінець рядка режиму додається літера b, наприклад “rb”, “wb”, “ab”, або, для смішаного режиму “ab+”, “wb+”, “ab+”. Якщо замість b додати літеру t, то файл буде відкриватися в текстовому режимі. В новому стандарті Сі (С11, 2011) можна додати літеру x, яка означає, що функція fopen для запису повинна завершитися з помилкою, якщо файл вже існує.

Як бачимо, режими відкривання файлу можуть бути доповнені специфікатором +, який означає режим **виправлення** – відкриття файлу одночасно і для запису, і для читання, наприклад:

FILE *lst;
lst=fopen(“D:\TC\file.txt”,”r+”);

Дані команди забезпечують відкриття вже існуючого текстового файлу **D:\TC\file.txt** як для читання, так і для запису.

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

int main() {
//З допомогою змінної file будемо мати доступ до файлу
FILE *file;
//Відкриваємо текстовий файл з правами на запис
file = fopen("C:/c/test.txt", "w+t");
//Пишемо в файл
fprintf(file, "Hello, World!");
//Закриваємо файл
fclose(file);
}
```

Відкриття файлу може відбутися з успіхом або з неуспіхом. У випадку успіху функція fopen створює в пам'яті службову структуру даних - потік, зв'язаний з потрібним файлом на диску, та повертає вказівникна цей потік. Цей вказівник потрібно зберегти в деякій змінній та потім використовувати в усіх операціях введення-виведення (див. приклад вище).

Неуспіх при спробі відкрити файл може виникнути з таких причин:

- для імені файлу, якого не існує на диску, задано режим r або r+;
- програма намагається відкрити для запису файл, для якого операційна система дає доступ лише для читання;
- програма відкрила надто багато файлів і вичерпала ліміт дозволених ресурсів;
- тощо.

В усіх таких випадках, коли замовлений файл в замовленому режимі відкрити неможливо, функція **fopen** повертає значення NULL.

Поведінка програми при неуспішному відкритті файлу та ще в багатьох схожих ситуаціях заслуговує на окремий розгляд. Гарно написана програма повинна бути стійкою до помилок, що можуть виникати під час виконання, через обставини, які програмісту та його програмі непідконтрольні. Скажімо, якщо програма запитує ім'я файлу у користувача, цілком може статися, що користувач помилково введе ім'я файлу, який насправді не існує. Або програма намагається записати дані у файл, а наявні у користувача права доступу забороняють запис.

Звичайно ж, програма повинна розпізнати таку нештатну ситуацію та коректно обробити її. Наведемо кілька типових рішень. В найпростішому випадку програма, побачивши, що файл відкрити неможливо, просто завершує свою роботу, друкуючи на екран повідомлення.

Зауважимо, що файл також бажано закрити по закінченню роботи за допомогою функції **fclose().**

Нагадаємо, що функція main повинна повертати ціле число, причому 0 символізує, що програма відпрацювала успішно, а будь-яке інше число означає, що програма завершилася через помилку.

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

#define ERROR_FILE_OPEN -3

int main() {
        FILE *output = NULL;
        int number;
        output = fopen("D:/c/output.bin", "wb");
        if (output == NULL) {
            printf("Error opening file");
            getchar();
        exit(ERROR_FILE_OPEN);
        }

        scanf("%d", &number);
        fwrite(&number, sizeof(int), 1, output);
```

```
        fclose(output);
}
```

Наведений вище фрагмент працює за доволі примітивною логікою, завершуючи програму у разі невдачі. Бажано зробити програму більш гнучкою: у випадку невдачі вона пропонує користувачу ввести інше ім'я файлу або, якщо користувач бажає, завершити роботу.

```
# include <stdio .h>
int main () {
char fileName [ 80 ];
FILE *f;
do {
  printf (" Введіть ім 'я файлу або крапку : " );
  scanf ( "%s", fileName );
  if( strcmp ( fileName , "." ) == 0 )
     return 0;
  f = fopen ( fileName , "r" );
} while ( f == NULL );
/* далі нормальна обробка файлу */
. . .
fclose ( f );
}
```

Читання з файлу

Якщо текстовий файл відкрито в режимі, що допускає читання, то прочитати з нього довільні дані можна за допомогою функції **fscanf**. Це майже повний аналог функції **scanf**. Перший аргумент – вказівник на потік, другий аргумент - форматний рядок, що містить будь-яку кількість специфікаторів формату, решта аргументів (їх кількість відповідає кількості специфікаторів) - покажчики на змінні, в які треба розмістити результати розбору прочитаного тексту.
Формат команди:
int fscanf(const char * filename, const char * format,char* buffer);
тобто
int fscanf (вказівник_на_файл,рядок форматування, перелік змінних);
// fscanf ("Імя файлу", "формат як в scanf", змінні через амперсант);

Функція **fscanf** намагається читати символи з потоку та співставляти їх з форматним рядком так само, як функція **scanf** розбирає послідовність символів, введену з клавіатури.
Функція повертає число успішно співставлених специфікаторів.
Нехай, наприклад, дано файл з іменем data.txt з таким вмістом:

Розглянемо програму (щоб не засмічувати розгляд, в тексті програми пропущено перевірку на помилку відкриття)

```
 # include <stdio .h>
 # define LEN 256
int main () {
        FILE *f;
        int m, n;
        double dt;
        char s[ LEN ];
        f = fopen (" data . txt ", "r");
        n = fscanf (f , "%d %lf %s", &m, &dt , s);
        printf (" Прочитано %d значень :\n", n);
        printf (" Ціле %d, дійсне %lf , рядок %s\n", m, dt , s);
         fclose ( f );
}
```

Ця програма успішно відкриє файл для читання. Функція **fscanf** прочитає з файлу символи та співставивши їх зі специфікатором ¾%d¿, перетворить їх на ціле число 15 та запише це число в змінну m. Точно таким же чином функція прочитає наступні знаки та занесе в змінну dt значення 37; 511. Далі, обробляючи специуікатор ¾%s¿, функція буде намагатися знайти у файлі послідовність символів до першого роздільника (пробіла, табуляції або переходу на новий рядок). Таким чином, функція scanf прочитає слово ¾князь¿, бо за ним йде пробіл, занесе його у масив s, приписавши до кінця нуль-символ кінця рядку. Решту тексту з файлу програма не прочитає взагалі (якби програма далі вводила ще дані цього файлу, вона прочитала б залишок, починаючи зі слова ¾Святослав¿).
Оскільки функція fscanf успішно співставила всі три специуікатори, вона поверне значення 3, яке і буде присвоєно змінній n. Тому програма надрукує на екран такий результат:
Прочитано 3 значень :
Ціле 15, дійсне -37.511 , рядок князь

Для введення текстового рядка, що містить пробіли та табуляції, слугує спеціальна функція fgets. Вона має такий прототип:
char * fgets (char *s, int n, FILE *f);
З прототипу видно, що функція має три аргументи: вказівникна символьний масив, в якому буде розміщено прочитаний з файлу текстовий рядок, ціле число границю довжини рядка, та вказівникна потік у файл, з якого треба читати рядок. Функція повертає покажчик на ту ж саму область пам'яті s, в яку розміщено прочитаний рядок. Функція читає текст з файлу, літера за літерою, поки не зустріне символ кінця рядка, кінець файлу, або поки не прочитає n-1 символ.

Замість функції fgets можна використовувати fscanf, але потрібно помятати, що вона зчитує дані лише до першого пробілу.
fscanf(file, "%127s", buffer);
Також, замість повторного відкриття та закриття файлу можна скористатися функцією freopen, яка «перевідкриавє» файл з новими правами доступу.

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

int main() {
	FILE *file;
	char buffer[128];
	file = fopen("C:/c/test.txt", "w");
	fprintf(file, "Hello, World!");
	freopen("C:/c/test.txt", "r", file);
	fgets(buffer, 127, file);
	printf("%s", buffer);
	fclose(file);
}
```

Функції роботи з файлами на диску

При переході від операцій запису до операцій читання часто необхідно застосовувати функцію **fflush,** яка забезпечує запис ще не збереженої інформації із буфера в файл. Вона має наступний формат:
int fflush (вказівник_на_файл);
Функція повертає EOF у разі помилки, та 0 в разі успішного виконання операції.
Після закінчення роботи з файлом його необхідно закрити командою **fclose**, яка має наступний формат:
int fclose(вказівник_на_файл);
Для видалення файла на диску слугує функція **remove**. Вона має наступний формат:
int remove(“ім’я_файлу”);
Для зміни імені файлу на диску є функція **rename**, яка має наступний формат:
int rename(“старе_ім’я_файлу”, “нове_ім’я_файлу”);
У разі успішності виконання функцій **remove, rename** вони повертають 0, та ненульове значення у іншому випадку.

Виведення/ Запис у файл

Форматоване виведення-введення даних у текстовий файл здійснюється за допомогою функцій **fprint** та **fscanf** віповідно, які мають наступний формат:
int fprintf (вказівник_на_файл,рядок форматування, перелік змінних);

Рядок форматування вміщує об’єкти трьох типів:

- -звичайні символи, які виводяться на екран;
- -специфікації перетворення, кожна з яких викликає виведення на екран значення чергового аргументу зі списку аргументів;
- -керуючі символи (для початку з нового рядка, табуляції, звукового сигналу та ін.).

Специфікація розпочинається символом **%** і закінчується символом, який задає перетворення. Між цими знаками може стояти:
- знак мінус “**-**”, який вказує, що параметр при виведенні на екран повинен вирівнюватися по лівому краю. Інакше - по правому;
- рядок цифр, який задає розмір поля для виведення. Крапка, яка відділяє розмір поля від наступного рядку цифр, що визначає розмір поля для виведення розрядів після коми для типів **float** та **double**;
- символ довжини **l**, який вказує на тип **long**;

Далі вказується символ типу інформації виведення (перетворення):
d- десяткове число;
o- вісімкове число;
x- шістнадцяткове число;
c- символ (тип char);
s- рядок символів (string);
e- дійсне число в експоненціальній формі;
f- дійсне число (float);
g- використовується як e і f, але виключає виведення на екран незначущих нулів;
u- беззнаковий тип (unsigned);
p- вказівник (pointer).
Якщо після **%** записано не символ перетворення, то він виводиться на екран.
Керуючі символи:
\a- для короткочасної подачі звукового сигналу (alarm);
\b- для переведення курсору вліво на одну позицію (back);
\n- для переходу на новий рядок (new);
\r- для повернення каретки (курсор на початок поточного рядка) (return);
\t- для горизонтальної табуляції (tabulation);
\v- для вертикальної табуляції (vertical).

Приклад. Відкрити файл для запису. Записати у файл матрицю з 10 рядків і 10 стовпців.

```
#include <stdio.h>
int main(){
  int i,j;
  FILE *lds;
   lds=fopen("epa.txt","w");
/*Відкриття файлу на диску для запису. Якщо він не існує, то створюється автоматично*/
for(i=1;i<=10;i++){
```

```
            for(j=1;j<=10;j++){
                  fprintf(lds,"%d%c",i+j-1,((j==10)?'\n':' '));}
            }
/*Запис даних до файлу: Перший аргумент – вказівник на файл, другий і третій такі ж як і для команди printf */
fprintf(lds,"\n");
fclose(lds); /*Закриття файлу*/
}
```

Сивольний запис у файл

Для деяких задач буває зручно вводити файл посимвольно.
Для цього слугує функція fgetc з таким прототипом:
int fgetc (FILE *f);
Для посимвольного введення та виведення даних у файл використовуються функції **fgetc** та **fputc** відповідно, які мають наступний формат:
int fgetc(вказівник_на_файл);
int fputc(вказівник_на_файл);
Функція **fgetc** повертає наступний символ із файла, на який вказує відповідний вказівник, або **EOF** в разі помилки. Функція **fputs** записує символ у файл, на який вказує відповідний вказівник, та повертає записаний символ, або **EOF** у випадку помилки.
Для введення-виведення рядків до файлу слугують функції **fgets** та **fputs**, які мають синтаксис, аналогічний **fgetc, fputc.**

Введення, виведення та повідомлень про помилки, stdin, stdout, stderr.

Для того, щоб засвоїти матеріал цього розділу, треба добре зрозуміти особливу ідеологію, що лежить в основі роботи з пристроями в операційній системі UNIX та у мові С. Основний принцип полягає в тому, що кожен приєднаний до комп'ютера пристрій (текстовий дисплей, клавіатура, принтер, модем, звукова плата) розглядається як своєрідний уявний файл. Наприклад, введення символів з клавіатури це те ж саме, що читання даних з уявного файлу, пов'язаного з клавіатурою, виведення на екран виглядає як запис у пов'язаний з екраном файл.
Ця надзвичайно вдала ідея дозволяє в однаковий спосіб обробляти введення з дискового файлу та з пристрою, такого, як клавіатура чи модем. Величезна вигода полягає в тому, що програма може не замислюватися над тим, звідки насправді беруться дані, з диску чи з пристрою у один і той самий текст програми підходить для обох випадків.
В заголовочному файлі stdio.h оголошено спеціальні змінні, в яких містяться покажчики на стандартні потоки введення-виведення. Найголовніші з них:

- **stdin** стандартний потік введення, зв'язаний з клавіатурою;
- **stdout** стандартний потік для виведення звичайної інформації, зв'язаний з дисплеєм;
- **stderr** стандартний потік, також зв'язаний з дисплеєм, але призначений для виведення повідомлень про помилки.
- ***stdprn*** стандартний потік виведення, пов'язаний з принтером.

Функція **printf** насправді виводить символи в стандартний потік виведення stdout, а функція введення **scanf** бере символи зі стандартного потоку введення stdin. Таким чином, наступні два оператори роблять в точності одне й те саме

printf (" Древляни , поляни , волиняни \n");
fprintf (stdout , " Древляни , поляни , волиняни \n");

Наступні два оператори також:

scanf ("%d", &x);
fscanf (stdin , "%d", &x);

Наведемо деякі корисні поради, пов'язані з таким підходом до обробки файлів.

Для друку повідомлень про помилки під час роботи програми призначено потік stderr.

ним і треба користуватися в гарно написаній програмі замість того, щоб виводити такі домлення функцією printf (в такому випадку повідомлення пішли б в інший потік st

Приклад:

```
 int n, *p;
 printf (" Кількість елементів ");
scanf ( "%d", &n );
p = (int *) malloc ( n * sizeof (int ) );
if( p == NULL ) {
fprintf ( stderr , "Не вистачає пам 'яті \n");
exit ( -1 );
}
 /* нормальна робота */
 . . .
```

Тут повідомлення для користувача, яке супроводжує нормальну роботу програми (на кількість елементів масиву), виводиться функцією printf і потрапляє у потік std повідомлення про помилку (нестача пам'яті) виводиться у спеціальний потік для помилок(хоча обидва ці потоки зрештою пов'язані з одним і тим самим дисплеєм).

Робота з бінарними файлами

Двійкові файли, на відміну від текстових, можуть містити такі коди, які не допускають відображення на екрані. Крім того, інформація у текстових файлах структурується розбиттям на рядки, пробілами між словами, табуляціями, структуру тексту видно людським оком.

Двійковий файл, навпаки, це просто послідовність байтів, не призначена для зручності читання людиною, двійковий файл може обробляти лише програма, яка «знає», який сенс має той чи інший байт.

Для обробки двійкових файлів використовуються ті ж потоки (змінні типу покажчиків на структуру **FILE**), що і для текстових файлів (див. попередню главу). Для відкриття двійкового файлу використовується та ж функція **fopen**, а для закриття функція **fclose**. *Відрізняються лише функції введення та виведення.*

Перш ніж починати описувати ці функції, треба зробити кілька вступних зауважень.

При роботі з файлами дуже часто доводиться мати справу з числами, сенсом яких є розмір файлу чи його частини. Звичайно ж, розмір вимірюється завжди цілим числом (причому невід'ємним), і можна було б скористатися звичайним

типом int. Але створювачі мови вирішили виділити цей тип окремою назвою, щоб підкреслити його особливе призначення:
size_t – це просто інше ім'я, запроваджене через **typedef**, для довгого беззнакового цілого типу.
При обробці текстових файлів функції введення спочатку читають з файлу символи, а потім розглядають їх як, наприклад, десятковий запис цілого числа і розміщують відповідне значення в пам'яті. В двійковому файлі числа зберігаються вже не в десятковому записі, а в такому ж точно вигляді, в якому вони існують в оперативній пам'яті машини. Іншими словами, записати інформацію до двійкового файлу значить перенести на диск точну копію вмісту деякої області оперативної пам'яті, байт за байтом.
В основу обробки двійкових файлів покладено таку ідею: обмін даними між пам'яттю та файлом здійснюється блоками однакового розміру. За одну операцію виведення можна взяти з пам'яті та скопіювати на диск певну кількість блоків, за одну операцію введення можна, навпаки, кілька блоків прочитати з файлу та записати в пам'ять. Важливо, що введення-виведення здійснюється лише цілими блоками (неможливо, скажімо, ввести півтора блоки).
Для введення-виведення даних до бінарних файлів застосовуються функції **fwrite** та **fread** відповідно.
Важливо, що блоки, які записуються у файл, повинні стояти у пам'яті підряд. Також і при читанні даних блоки, прочитані з файлу, розташуються у пам'яті підряд. Для запису даних до двійкового файлу призначена функція fwrite, яка має чотири аргументи:

1. **void *p** – вказівник на те місце в оперативній пам'яті, де починається послідовність блоків даних, яку треба записати у файл;
2. **size_t b** – довжина в байтах одного блоку;
3. **size_t n** – число блоків;
4. **FILE *f** – вказівник на потік: до якого файлу записати дані.

Функція бере з пам'яті, починаючи з місця, на яке вказує вказівник, n блоків даних, кожен розміром b байт, та записує їх у файл, пов'язаний з потоком f.
Функція повертає ціле значення число блоків, які їй вдалося записати. При успішній роботі, звичайно ж, це число повинно дорівнювати n, менше значення повинно свідчити про помилку у процесі запису у файл (наприклад, перевовнення диску). В підсумку, функція має прототип:
size_t fwrite (void *p, size_t b, size_t n, FILE *f);
або
int fwrite(вказівник_на_масив, розмір_об’єкта, кількість_об’єктів , вказівник_на_файл);
Функція забезпечує запис об’єктів із масиву, на який вказує **вказівник_на_масив**, кількість об’єктів визначається як **кількість_об’єктів,** розміром **розмір_об’єкта** у файл, на який вказує **вказівник_на_файл**. Функція повертає кількість успішно записаних об’єктів.

Для введення даних з двійкового файлу призначена функція fread. Вона має ті ж аргументи, що й функція fwrite, але передає дані в протилежному напрямку: з потоку f читає n блоків даних, кожен розміром b байт, та розміщує їх в пам'яті, починаючи з місця, на яке вказує вказівник.
Вони мають наступний формат:
size_t fread (void *p, size_t b, size_t n, FILE *f);
або
int fread (вказівник_на_масив, розмір_об'єкта, кількість_об'єктів , вказівник_на_файл);
Функція забезпечує зчитування об'єктів у масив, на який вказує **вказівник_на_масив**, кількість об'єктів визначається як **кількість_об'єктів,** розміром **розмір_об'єкта** із файла, на який вказує **вказівник_на_файл**. Функція повертає кількість успішно записаних об'єктів.

Розглянемо інший приклад: нехай програма також записує до файлу числа, але не весь масив за одну операцію, а по одному:

```
 # include <stdio .h>
 # define N 5
int main () {
double w[ N ] = { 2.0 , 1.4142 , 1.1892 , 1.0905 , 1.0443 };
 char fileName [] = " data . dat ";
FILE * out ;
int i;
out = fopen ( fileName , "w" );
 for ( i = 0; i < N; ++i )
fwrite ( &(w[i]), sizeof(double), 1, out );
fclose ( out );
}
```

Тут функція запису викликається в циклі 5 разів, при кожному виклику вона записує у айл вміст чергового елементу масиву w: аргументи означають, що записувати треба дані, очинаючи з тієї адреси, де лежить ця змінна, записати треба один блок такого розміру, який ймає в пам'яті дійсне число.
Тепер розглянемо програму, яка читає та роздруковує масив дійсних чисел з файлу data.dat¿, створеного попередньою програмою.

```
# include <stdio .h>
# define N 5
int main () {
        double w[ N ];
        char fileName [] = " data . dat ";
        FILE *inFile;
        int k, i;
        inFile = fopen ( fileName , "w" );
        k = fread ( w, sizeof ( double ), N, inFile );
```

```
        printf ( "З файлу прочитано %d чисел ", k );
        for ( i = 0; i < k; ++i ){
                printf ( "%lf\n", w[i] );
                }
        fclose (inFile );
}
```

Головним місцем програми є виклик функції **fread**, яка з inFile читає N (тобто 5) блоків даних, кожен такого ж розміру, як дійсне число, а ці дані розміщує в пам'яті там, де починається масив w. Отже, програма читає з inFile 5 чисел та присвоює їх елементам масиву. Значенням змінної k при цьому стає кількість вдало прочитаних блоків даних (тобто чисел).

Розглянемо також програму, яка читає та роздруковує числа з файлу один за одним:

```
# include <stdio .h>
int main () {
double x;
char fileName [] = "data.dat";
FILE *outFile;
int k =0;
in = fopen ( fileName , "r" );
while ( ! feof (outFile) ) {
        fread ( &x, sizeof ( double ), 1, in );
        printf ("%lf\n", x);
         ++k;
 }
printf (" всього %d чисел \n", k);
fclose (outFile );
}
```

Ця програма читає числа по одному, одне число за одну операцію. Аргументи функції fread означають: взяти з файлу (потоку) outFile один блок даних такого ж розміру, як дійсне число, та покласти в оперативну пам'ять, в ту комірку, де розташована змінна x. Цикл означає, що програма буде повторювати вказану операцію доти, поки не закінчиться файл, а кожне значення, прочитане з файлу та занесене до змінної , буде друкуватися на екрані.

Розгялнемо роботу з файлами при використанні структур у якості параметрів, що треба прочитати або записати:

```
#define  SURNAME_LEN 30
typedef struct {
        char name[SURNAME_LEN]; int kurs;
        int prog_lab, prog_theory;
 } Student;

int main()
```

```
        char fileName [SURNAME_LEN];
        FILE * out ;
        Student s;
        int ans , n =0;
        printf ("Ім 'я файлу ? ");
        fgets ( fileName , SURNAME_LEN, stdin );
        out = fopen ( fileName , "w" );
        do {
                printf ("Ім 'я? "); printf (" Прізвище ? ");
                fscanf (s. surname , SURNAME_LEN , stdin );
                printf (" Оцінки з теорії та лаб . робіт ? ");
                scanf ("%d%d", &(s. prog_theor ), &(s. prog_lab ));
                fwrite ( &s, sizeof (Student), 1, out );
                 ++n;
                printf (" Продовжити (1 так , 0 ні )? ");
                 scanf ("%d", & ans );
        } while ( ans );
        printf ("У файлі %d записів \n", n);
        fclose ( out );
}
```

як видно, блоком даних є структура типу Student, за кожну операцію до файлу виводиться рівно один блок вміст змінної s. Після завершення програми дані у файлі будуть сформовані так:

Щоб проілюструвати читання структур з файлу в бінарному варіанті, наведемо програму, яка читає базу даних по студентах, сформовану попередньою програмою, та роздруковує її її вміст на екрані у вигляді таблиці з трьох колонок.

```
# include <stdio .h>

# define FNAME_LEN 80
# define SURNAME_LEN 20

 typedef struct tag_student {
char surname [ SURNAME_LEN ];
int prog_theor ;
 int prog_lab ;
} Student;

int main () {
char fileName [ FNAME_LEN ];
FILE *in;
Student s;
```

```
int n =1;
printf ("Ім 'я файлу ? ");
fgets ( fileName , FNAME_LEN , stdin );
in = fopen ( fileName , "r" );
while ( ! feof (in) ) {
  fread ( &s, sizeof (Student), 1 );
  printf ("%3d %30 s %10 d %10 d\n", n, s. surname , s. prog_theor , s. prog_lab );
 ++n;
}
fclose ( in );
}
```

Пошук у бінарному файлі: безпосередній доступ

Обробка текстових файлів частіше за все полягає у послідовному, літера за літерою, читанні одного файлу та такому ж послідовному записі результатів в інший файл. При обробці двійкових файлів у багатьох практичних задачах виникає потреба читати чи записувати файл не підряд, а в довільному порядку, час від часу переміщуючись по ньому вперед і назад.

Уявімо, що файл це довга стрічка, поділена на комірки, в кожній з яких може зберігатися один байт. По стрічці переміщується маркер читання-запису. При звичайних (послідовних) операціях читання та запису маркер, прочитавши чи записавши комірку навпроти якої стоїть, переміщується на одну комірку вперед. Разом з тим, в мові С існують функції, які дозволяють рухати маркер на довільну відстань вперед та назад по файлу. якщо встановити голівку на певну комірку, наступна операція читання чи запису буде відноситися саме до цієї комірки.

Функція fseek доволі універсальна: вона дозволяє встановити маркер на n-й байт, рахуючи від початку файла, на n-й байт від кінця файлу, або перемістити маркер на n байтів (вперед чи назад) відносно її поточного місця. Функція має три аргументи:

- вказівникна потік в якому файлі встановлювати маркер;
- ціле число на яку відстань переміщувати маркер;
- ціле число, що означає режим переміщення маркер (рахувати відстань від початку, від кінця файлу, або від поточної позиції маркеру).

Часто при роботі з файлами корисно використовувати по елементний доступ до файлу. Для цього застосовується функція **fseek**, що має формат:

Int fseek(вказівник_файла, зміщення, початкове_значення);

Функція встановлює позицію із **зміщенням** відносно початкового_значення у файлі, який визначається **вказівником файла. Початкове значення** може приймати наступні значення:

SEEK_SET – зміщення відносно початку файла;

SEEK_CUR – зміщення відносно поточного положення;

SEEK_END – зміщення відносно кінця файла;
Функція **ftell** повертає поточну позицію у файлі, та має наступний формат: За допомогою функцій **fseek** та **ftell** легко дізнатися розмір файлу. Для цього достатньо спочатку поставити голівку на кінець файлу, а потім дізнатися її поточну позицію:

```
FILE *f;
long l;
char fileName = "data.dat ";
f = fopen ( fileName , "r" );
fseek ( f, 0L, SEEK_END );
l = ftell ( f );
printf ( " Довжина файлу %ld байтів \n", l );
```

long ftell(вказівник_на_файл);
Приклад 2. Продемонструвати роботу із бінарними файлами.

```
#include <stdio.h>

int main(void){
        const unsigned n=10;
        int a[10];
        FILE *pfile;
        int i,j,pos,start;
        int * pstart;
        pstart=&start; /* вказівник вказує на змінну start */
        clrscr();
        printf("\nFilling vector with numbers...\n\n a=[ ");
        for(i=0;i<10;i++){
        a[i]=i+1;
        printf("%d ",a[i]);

        printf("]\n\nCreating binary file epa.dat for editing...");
        pfile=fopen("epa.dat","w+b"); /*створення нового бінарного файлу для
        редагування */
        j=fwrite(a,sizeof(int),n,pfile); /* запис елементів вектора a у бінарний файл
        */
        if(j<n){
                printf("\n\nAn error occured. Only %d of %d elements was written",j,n);
        }
        else printf("\n\nFile was filled with %d elements successfully",j);
        fflush(pfile);
        /* перед зчитуванням даних в режимі редагування обов'язково дописуємо
        вміст буфера у файл */
        printf("\n\nEnter the number of element you want to read from bin file ");
        scanf("%d",&j);
        fseek(pfile,(j-1)*sizeof(int),SEEK_SET);
```

```
/*перехід до позиції необхідного елемента*/
pos=ftell(pfile);
/*Визначення позиції у файлі*/
fread(pstart,sizeof(int),1,pfile);
printf("\n\n SEEK_SET: %dth element position is %d, element is %d",j,pos,*pstart);
printf("\n\nPress any key to exit");
fclose(pfile); /* закриття файла */
}
```

Підсумок

Заголовок <stdio.h> надає загальну підтримку операцій з файлами та надає функції з вузькими можливостями введення / виводу символів.
Заголовок <wchar.h> надає функції з широкими символами можливостями введення / виводу символів.

Таблиця 5.2

Функції роботи з файлами

В <stdio.h> - загальна робота з файлами	
fopen, fopen_s(C11)	Відкриває файл (функція)
freopen, freopen_s(C11)	Перевдкріває файл (функція)
fclose	Закриває файл
fflush	Сінзронізує вивід з поточним, очищає буфер(function)
setbuf	Встановлює буфер для потоку
setvbuf	Встановлює буфер та його розмір для потоку
В <wchar.h>	
fwide (C95)	Перключає буфер широких символів I/O та вузьких символів I/O (function)
Введення, виведення	
В <stdio.h>	
fread	Читає з файлу
fwrite	Записує в файл
fgetc, getc	Читає символ
fgets	Читає рядок
fputc, putc	Записує символ
fputs	Записує рядок
getchar	Читає символ з **stdin** (function)

gets(до C11), gets_s(з C11)	Читає рядок з stdin (function)
putchar	Записує символ у stdout (function)
puts	Записує рядок у **stdout**(function)
ungetc	Повретає символ обратно в буфер

Широкі символи

В <wchar.h>

fgetwc, getwc(C95)	Читає широкий символ з потоку(function)
fgetws(C95)	Читає широкий рядок з потоку (function)
fputwc, putwc(C95)	Записує широкий символ в поток (function)
fputws(C95)	Записує широкий рядок в поток
getwchar(C95)	Читає широкий смивол з stdin (function)
putwchar (C95)	Записує широкий символі до **stdout** (function)
ungetwc(C95)	Повиертає широкий рядок в потік

Форматоване введення/виведення

В <stdio.h>

scanf, fscanf, sscanf, scanf_s(C11), fscanf_s(C11), sscanf_s (C11)	Читає форматований рядок з **stdin**, потоку або буферу(function)
vscanf(C99), vfscanf(C99), vsscanf(C99), vscanf_s(C11), vfscanf_s(C11), vsscanf_s (C11)	Читає форматований рядок з **stdin**, потоку або буферу використовуючи форматований список змінних (function)
printf, fprintf, sprintf, snprintf(C99), printf_s(C11), fprintf_s(C11), sprintf_s(C11), snprintf_s (C11)	Друкує форматований ввод у **stdout**, потік або буфер(function)
vprintf, vfprintf, vsprintf, vsnprintf(C99), vprintf_s(C11), vfprintf_s(C11), vsprintf_s(C11), vsnprintf_s (C11)	Друкує форматований ввод у **stdout**, потік або буфер(function використовуючи форматований список змінних (function)

Wide character

Defined in header <wchar.h>

wscanff(C95),wscanfs(C95),wscanf(C95),wscanf_s(C11),fwscanf_s(C11),s	Читає широкий форматований рядок з **stdin**, файлу або буферу(function)

wscanf_s (C11)	
vwscanf(C99),vfwscanf(C99),vswscanf(C99), vwscanf_s(C11),vfwscanf_s(C11),vswscanf_s (C11)	Читає широкий форматовіний рядок з **stdin**, потоку або буфер через список змінних(function)
wprintf(C95),fwprintfs(C95),wprintf(C95),wprintf_s(C11),fwprintf_ss(C11),wprintf_s(C11),snwprintf_s (C11)	Друкує форматований рядок в **stdout**, поток або буфер(function)
vwprintf(C95),vfwprintf(C95),vswprintf(C95),vwprintf_s(C11),vfwprintf_s(C11),vswprintf_sм,vsnwprintf_s (C11)	Друкує форматований рядок **stdout**, поток або буфер використовуючи списко змінних(function)

Позиції в файлі

В <stdio.h>

ftell	Повертає поточну позицію маркеру (function)
fgetpos	Повертає позицію у файлах(function)
fseek	Ставить маркер на потрібну позицію(function)
fsetpos	Ставить маркер на фіксовану позицію у файлі (function)
rewind	Повертає маркер на початок файлу(function)

Обробка помилок

В <stdio.h>

clearerr	Помилка очистки буфера(function)
feof	Перевіряє чи ми на кінці файлу(function)
ferror	Промила у файлі(function)
perror	Рядок з поточної помилки у **stderr** (function)

Дії над файлами

В <stdio.h>

remove	Знищує файл(function)
rename	Переіменовує файл(function)
tmpfile, tmpfile_s(C11)	Вказівник на поточний файл(function)
tmpnam, tmpnam_s(C11)	Повертає унікальне імя файлу(function)

Типи

В <stdio.h>

Тип	Означення
FILE	Тип для контролю С-шного I/O потоку

fpos_t Тип, що визначає позицію та стан маркера у файлі

Макроси

В <stdio.h>

Stdin,stdout,stderr	Вираз типу FILE* ассоційований вхідним потоком консолі, Вираз типу FILE* ассоційований з вихідним потоком консолі, Вираз типу FILE* ассоційований з вихідним потоком консолі для помилок (macro constant)
EOF	Цілий константний вираз (constant expression of type int) для кінця файлу(macro constant)
FOPEN_MAX	Максимальнва кількість файлів, що може бути відкрита одночасно (macro constant)
FILENAME_MAX	Розмір потрібний рядку байтів для найбільшого за розміром імені файлу(macro constant)
BUFSIZ	Розмір буферу визначений setbuf() (macro constant)
_IOFBF,_IOLBF,_IONBF	аргумент setvbuf() позначающій fully buffered I/O аргумент setvbuf()позначающій line buffered I/O аргумент setvbuf()позначающій unbuffered I/O (macro constant)
SEEK_SET, SEEK_CUR, SEEK_END	аргумент fseek() – начало файлу; аргумент fseek() поточна позиція; аргумент fseek() – кінець файлу (macro constant)
TMP_MAX, TMP_MAX_S (C11)	Максимвльна кількість імен в tmpnam Максимвльна кількість імен в tmpnam_s (macro constant)
L_tmpnam, L_tmpnam_s(C11)	Розмір масиву char для результату tmpnam; Розмір масиву char для результату tmpnam_s (macro constant)

Посилання

1. C11 standard (ISO/IEC 9899:2011):
   7.21 Input/output <stdio.h> (p: 296-339)
   7.29 Extended multibyte and wide character utilities <wchar.h> (p: 402-446)
   7.31.11 Input/output <stdio.h> (p: 456)
   7.31.16 Extended multibyte and wide character utilities <wchar.h> (p: 456)
   K.3.5 Input/output <stdio.h> (p: 586-603)
2. C99 standard (ISO/IEC 9899:1999):
   7.19 Input/output <stdio.h> (p: 262-305)
   7.24 Extended multibyte and wide character utilities <wchar.h> (p: 348-392)
   7.26.9 Input/output <stdio.h> (p: 402)