**Стандартні бібліотечні функції**

*Бібліотека* язика *Сі* містить безліч функцій і *макроозначень*. Бібліотеки змінюються від системи до системи, але є *ядро* функцій (стандартна бібліотека).

Ці функції використовуються для:

* маніпулювання даними, їх перетворення і шифрування;
* визначення користувачами функцій зі змінним числом аргументів;
* динамічного управління *пам'яттю*;
* представлення показань системних годин в стандартних форматах дати і часу;
* отримання системної інформації.

Головними перевагами стандартних бібліотечних функцій *Сі* є мобільність і низькі витрати на супровід призначених для користувача додатків. Бібліотечні функції не схильні до частих змін, тому програми, в яких вони використовуються, легкі в супроводі. Деякі з цих бібліотечних функцій відповідають стандарту ***ANSI С*** - стандарту *Сі* Американського національного інституту, завдяки чому вони прийнятні для всіх систем, що відповідають цьому стандарту. Щоб скоротити витрати і час на розробку додатків, рекомендується використовувати бібліотечні функції *Сі* щоразу, коли це виявляється можливим.

Стандартні бібліотечні функції Сі оголошуються в наборі файлів-заголовків, які в UNIX-системах зазвичай розташовані в каталозі / usr / include. Наведемо бібліотечні функції ANSI C, певні в файлах-заголовків, перерахованих нижче:

<stdio.h>

<stdlib.h>

<string.h>

<memory.h>

<malloc.h>

<time.h>

<assert.h>

<stdarg.h>

<getopt.h>

<setjmp.h>

Крім зазначених, в більшості UNIX-систем є *файли заголовків*, які не визначені в ANSI C:

<pwd.h>

<grp.h>

<crypt.h>.

У цих *файлах заголовків* оголошуються функції, які допомагають отримати доступ до інформації про облікові записи користувачів і груп в UNIX-системах. У зазначених системах вони визначені в бібліотеці libc.a. Ці функції корисні для розробки додатків.

У файлі заголовків <stdio.h> оголошується тип даних FILE, який використовується в *Сі*-програмах для позначення *потокових файлів*, або просто **потоків**, тобто *файлів*, обмін з якими здійснюється за допомогою функцій потокового введення-виведення. Є також набір макрокоманд і функцій, призначених для маніпулювання потоковими файлами. Нижче наведені деякі з цих макрокоманд і функцій, які вже повинні бути знайомі з попередніх лекцій.

|  |  |
| --- | --- |
| **Потокова функція або макрокоманда** | **Призначення** |
| fopen | Відкриває *потік* для читання і (або) записи |
| fclose | Закриває *потік* |
| fread | Читає блок данних з *потоку* |
| fgets | Читає рядок тексту з *потоку* |
| fscanf | Читає відформатовані дані з *потоку* |
| fwrite | Записує блок даних в *потік* |
| fputs | Записує рядок тексту в *потоці* |
| fprintf | Записує відформатовані дані в *потік* |
| fseek | Переміщує *покажчик* читання або запису в *потоці* |
| ftell | Повертає поточну позицію в *потоці*, починаючи з якої буде виконана наступна операція читання або запису. Значення, що повертається - це кількість байтів зміщення відносно початку *потоку* |
| freopen | Повторно використовує покажчик потоку для посилання на новий *файл* |
| fdopen | Відкриває потоковий файл із зазначеним дескриптором |
| feof | Макрокоманда, яка повертає нульове значення, якщо в даному *потоці* виявлено символ кінця *файлу*, в іншому випадку - нульове значення |
| ferror | Макрокоманда, яка повертає нульове значення, якщо в даному *потоці* була виявлена ​​помилка або символ кінця *файлу*, в іншому випадку - нульове значення |
| clearer | Макрокоманда, яка скидає прапор наявності помилок в даному *потоці* |
| fileno | Макрокоманда, яка повертає дескриптор даного *потокового* файлу |

У заголовку <stdlib.h> оголошується набір функцій, що служать для перетворення даних, генерації випадкових чисел, отримання і установки змінних середовища shell, управління виконанням програм і виконання команд shell. Зазвичай ці функції оголошуються в заголовку <stdio.h>, але так як вони не включають в себе маніпулювання потоками, стандарт ANSI C групує їх в окремий заголовок.

У заголовку <string.h> оголошується набір функцій, призначених для маніпулювання символьними рядками.

У заголовку <memory.h> оголошується набір функцій, призначених для маніпулювання байтовим *потоком*. Ці функції схожі на строкові, але на відміну від них мають більш широке призначення і можуть використовуватися для маніпулювання несімвольнимі строковими об'єктами. Зокрема, дані функції можна застосовувати для ініціалізації, порівняння і копіювання об'єктів типу struct.

У заголовку <time.h> оголошується набір функцій, призначених для виклику системних параметрів часу. Вони можуть застосовуватися для визначення місцевого часу і дати, часу і дати в універсальному форматі *(UTC*), а також статистичних даних про використання процесами часу центрального процесора.

У заголовку <assert.h> оголошується макрокоманда, використовувана для перевірки деяких умов виконання процесу, які в нормальній ситуації завжди повинні бути істинні. Якщо все ж під час виконання процесу умова не виконується, то макрокоманда виводить повідомлення про помилку в стандартний потік помилок із зазначенням того рядка вихідного файлу, в якій порушується перевіряється умова. Після цього макрокоманда перериває процес.

У заголовку <setjmp.h> оголошується набір функцій, які дозволяють процесу викликати оператор переходу goto з однієї функції в іншу. Виклик Сі-оператора goto дозволяє процесу передати управління виконанням від одного оператора до іншого лише в рамках цієї ж функції. Функції, визначені в заголовку <setjmp.h> усувають це обмеження. Ці функції необхідно використовувати лише тоді, коли без них дійсно не можна обійтися. Наприклад, якщо помилку виявлено в рекурсивної функції, то є сенс повідомити про помилку, а потім виконати оператор переходу goto в основну функцію, тобто як би почати процес спочатку.

У заголовку <pwd.h> визначається набір функцій, призначених для отримання облікової інформації про користувачів.

У заголовку <grp.h> визначається набір функцій, призначених для отримання облікової інформації про групи, що міститься в UNIX- файлі / etc / group.

У заголовку <crypt.h> оголошується набір функцій, призначених для шифрування і дешифрування даних. Це дуже важливі функції, що забезпечують безпеку системи. Наприклад, файли для користувача паролів і системних даних, яким необхідна висока ступінь захисту, повинні бути зашифровані так, щоб жодна людина, що не має спеціального дозволу, не міг дізнатися, що вони з себе представляють. Більш того, щоб читати і змінювати ці об'єкти, уповноважені особи повинні знати секретні ключі дешифрування.

**Доступ до бібліотеки мови Сі**

Отримання доступу до бібліотеки залежить від системи. По-перше, є кілька різних місць розташування бібліотечних функцій, Наприклад, getchar () зазвичай задають як макроозначень в файлі stdio.h, в той час як strlen () зазвичай зберігається в бібліотечному файлі. По-друге, різні системи мають різні способи доступу до цих функцій. Ось три з них.

***Автоматичний доступ***

У багатьох великих системах UNIX ви тільки компілюєте програми. А доступ до більш загальним бібліотечних функцій виконується автоматично.

#### Включення файлу

Якщо функція задана як макроозначення, то можна директивою #include включити файл, який містить її визначення. Часто подібні функції можуть бути зібрані в відповідним чином названий заголовки. Наприклад, деякі системи мають файл ctype.h, що містить макроозначення, що задають тип символу (прописна буква, цифра и т.д.)

***Включення бібліотеки***

На деякому етапі компіляції або завантаження програми ви можете вибрати бібліотеку. Навіть система, яка автоматично контролює свою стандартну бібліотеку, може мати інші бібліотеки рідко вживаних функцій, і ці бібліотеки слід запитувати явно, вказуючи відповідний ознака під час компіляції.

Очевидно, ми не зможемо розглянути всі особливості всіх систем, але ці три приклади показують, що вас очікує!

**Зв'язок з файлами**

Один спосіб організації зв'язку програми з файлом полягає в використанні операцій перемикання <і>. Цей метод простий, але обмежений. Мова Сі надає і більш потужні методи зв'язку з файлами. Розглянемо використання функції fopen (), яка відкриває файл, потім застосовуються спеціальні функції введення-виведення для читання файлу або запису в цей файл і далі використовується функція fclose () для закриття файлу. Перш ніж досліджувати ці функції, коротко познайомимося з сутністю файлу.

Файл є частиною пам'яті, зазвичай на диску, зі своїм ім'ям. Ми вважаємо, що він містить деяку корисну інформацію. Для операційної системи файл складніший, але це системні проблеми, а не наші. Але ми повинні знати, що означає файл для програми на мові Сі. У пропонованих для обговорення функціях, які працюють з файлами, мова Сі розглядає файл як структуру. Ось типовий приклад, взятий з IBM-версії компілятора Lattice C:

struct\_iobuf {

char\*\_ptr; /\* поточний покажчик буфера \*/

int\_cnt; /\* поточний лічильник байтів \*/

char\*\_base; /\* базовий адреса буфера введення-виведення \*/

char\_flag; /\* керуюча ознака \*/

char\_file; /\* номер файла\*/

}

#define FILE struct\_iobuf /\* короткий запис \*/

Тут ми не збираємося розбиратися детально в цьому визначенні. Головне полягає в тому, що файл є структурою, і що коротке найменування шаблону - FILE. Багато систем використовують директиву typedef для встановлення цієї відповідності. Таким чином, програма, що має справу з файлами, буде використовувати тип структури FILE, щоб робити так.

Розглянемо приклад читання вмісту файлу, названого File, і виведення його на екран:

#include <stdio.h>

main( )

{

FILE \*in; /\* описуємо покажчик на файл \*/

int ch;

if ((in = fopen("File", "r") ) != NULL) {

/\* відкриваємо File для читання,

перевіряючи чи існує він \*/

/\* покажчик FILE посилається тепер на File \*/

while ((ch = getc(in)) != EOF)

/\* отримуємо символ з in \*/

putc(ch, stdout);

/\* посилаємо ch на стандартний висновок \*/

fclose(in); /\* закриваємо файл \*/

}

else

printf (" Файл не відкривається \"File\".\n");

}

Пояснимо роботу: fopen( ), fclose і використання функцій вводу-виводу файлу.

#### Відкриття файлу: fopen( )

Функцією fopen () керують три основних параметри. Перший - ім'я файлу, який слід відкрити. Він є і першим аргументом fopen (). У нашому прикладі це "File". Другий параметр описує, як повинен використовуватися файл:

" r " - *файл потрібно рахувати*,

" w " - *файл потрібно записати*,

" a " - *файл потрібно доповнити*.

" w+ " - новий *текстовий файл* відкривається для запису і наступних багаторазових виправлень. Якщо файл вже існує, то попередній вміст стирається. Наступні після відкриття файлу запис і читання з нього є допустимими в будь-якому місці файлу, в тому числі запис дозволена і в кінці файлу, тобто файл може збільшуватися.

" r+ " - існуючий текстовий *файл* відкривається як для читання, так і для запису в будь-якому місці *файлу*; проте в цьому режимі неможливий запис в кінець *файлу*, тобто неприпустимо збільшення розмірів *файлу*.

" a+ " - *текстовий файл* відкривається або створюється, якщо *файлу* немає, і стає доступним для змін, тобто для запису і для читання в будь-якому місці; при цьому на відміну від режиму "w +" можна відкрити існуючий *файл* і не знищувати його вміст; на відміну від режиму "r +" в режимі "a +" можна вести запис в кінець *файлу*, тобто збільшувати його розміри.

Деякі системи надають ще додаткові можливості, які ми тут не будемо розглядати. Використовувані коди є рядками, а не символьними константами. При застосуванні "r" відкривається існуючий *файл*. При двох інших застосуваннях теж буде відкриватися існуючий *файл*, але якщо такого файлу немає, він буде створений. **Якщо ви використовуєте** "w" **для існуючого файлу, то стара версія його стирається, і ваша програма починає записувати на чисте місце**. Третій параметр є покажчиком на файл. Це значення повертається самою функцією:

FILE \*in;

in=fopen("File","r");

Тепер in є *покажчиком на файл* " File ". З цього моменту програма посилається на файл за допомогою покажчика in, а не по імені File. ( *Файл* stdio.h містить рядок

FILE \*fopen( )

Якщо fopen( ) не здатна відкрити потрібний *файл*, вона повертає значення NULL, певне в stdio.h як 0.

#### Закриття файла: fclose( )

У нашому прикладі показано, як потрібно закривати *файл*:

fclose(in);

Аргумент функції є покажчиком на файл. Для більш серйозної програми потрібно дивитися, чи успішно закритий файл. Функція fclose( ) повертає значення 0, якщо *файл* закритий успішно, і EOF в іншому випадку.

***Текстові файли з буферизацією***

Функції fopen( ) та fclose( ) працюють з текстовими *файлами* з "буферизацией ". Під ***буферизацією*** ми розуміємо, що вводяться і виводяться дані запам'ятовуються в тимчасовій області пам'яті, званої **буфером**. Якщо буфер заповнився, вміст його передається в блок, і процес буферизації починається знову. Одна з основних завдань fclose () полягає в тому, щоб звільнити будь-які частково заповнені буфери, якщо *файл* закритий. Текстовим вважається *файл*, в якому інформація запам'ятовується у вигляді символів в коді ASCII або аналогічному. Текстовий *файл* відрізняється від виконуваного файлу, який зазвичай використовується для запам'ятовування кодів машинної мови.

#### Введення-виведення текстового файлу: getc( ), putc( )

Дві функції getc () і putc () працюють аналогічно функціям getchar () і putchar () (описаним в попередніх лекціях). Різниця полягає в тому, що ви повинні повідомити, який *файл* слід використовувати.

char ch;

ch=getchar( );

призначена для отримання символу від стандартного введення, а

ch=getc(in);

- для отримання символу від файлу, на який вказує in.

putchar(ch);

виводить символ на стандартний файл виведення.

putc(ch,t);

призначена для запису символу ch в файл, на який посилається покажчик t типу FILE.

#### Введення-виведення файлу: fprintf( ), fscanf( ), fgets( ), fputs( )

Всі функції введення-виведення, які ми використовували в попередніх лекціях, мають аналоги для введення-виведення файлу. Основна відмінність полягає в тому, що нам потрібно використовувати покажчик типу FILE, щоб повідомити функцій з яким файлом їм слід працювати. Подібно getc () і putc () ці функції використовуються після функції fopen (), що відкриває файл, і перед fclose (), що закриває його.

#### Функції fprintf( ) і fscanf( )

Ці функції введення-виведення працюють майже як printf () і scanf () (див. Лекцію 4), але їм потрібен додатковий аргумент для посилання на сам файл. Він є першим в списку аргументів. Приклад, який ілюструє звернення до цих функцій:

#include <stdio.h>

main( )

{

FILE \*fi;

int age;

fi=fopen("File","r"); /\* зчитування \*/

fscanf(fi,"%d",&age); /\* fi вказує на File \*/

fclose(fi);

fi=fopen("Data", "a"); /\* доповнення \*/

fprintf(fi,"Data is %d.\n",age);

/\*fi вказує на Data\*/

fclose(fi);

}

На відміну від getc( ) і putc( ) ці функції отримують покажчик типу FILE в якості першого аргументу.

#### Функція fgets( )

Ця функція має три аргументи, в той час як gets( ) має лише один. Приклад її використання:

/\* Програма зчитує файл рядок за рядком \*/

#include <stdio.h>

#define MAX 80

main( )

{

FILE \*f1;

char \*string[MAX];

f1=fopen("File","r");

while (fgets(string,MAX,f1) != NULL)

puts(string);

}

Ми розташували вводиться інформацію в символьному *масиві* string. Перший з трьох аргументів функції fgets( ) є покажчиком на розташування зчитуємого рядка. Другий аргумент містить максимальну довжину зчитуємого рядка. Функція припиняє роботу після зчитування символу нового рядка або після зчитування символів загальним числом MAX-1, в залежності від того, що станеться раніше. У будь-якому випадку нуль-символ ' \0 ' додається в самий кінець рядка. Третій аргумент вказує на файл, який буде читатися. Різниця між gets( ) і fgets( ) полягає в тому, що gets( ) замінює символ нового рядка на ' \0 ', в той час як fgets( ) зберігає символ нового рядка. Подібно gets( ) функція fgets( ) повертає значення NULL, якщо зустрічає символ EOF . Це дозволяє нам перевірити, чи досягли ми кінця файлу.

#### Функція fputs( )

Ця функція схожа на функцію puts( ). Оператор

l=fputs("Строка", fi);

|  |  |
| --- | --- |
| **Код** | **Положение в*****файлі*** |
| 0 | початок *файлу* |
| 1 | поточна позиція |
| 2 | кінець файлу |

Передає рядок " Строка " в файл, на який посилається вказівник fi типу FILE. Звичайно, спочатку потрібно відкрити файл за допомогою *функції* fopen( ).

l є цілим числом, яке встановлюється в EOF, якщо fputs( ) зустрічає EOF або помилку. Ця функція не ставить завершальний символ ' \0 ' в кінці копіюючогося рядка. На відміну від puts функція fputs () не встановлює символ нового рядка в її висновок.

#### Функція fseek( )

Функція fseek () дозволяє нам обробляти файл подібно масиву і безпосередньо досягати будь-якого певного байта у файлі, відкритому функцією fopen (). fseek () має три аргументи і повертає значення типу int.

Покажемо на прикладі роботу fseek( ):

/\* використання fseek( ) для друку вмісту файлу \*/

#include <stdio.h>

int main(int number, char \*names[])

{

FILE \*fp; long set = 0L;

if(number<2)

puts("Введіть ім'я файлу в якості аргументу.");

else {

if ((fp=fopen(names[1],"r")) == 0)

printf("Не можна відкрити %s\n",names[1]);

else {

while(fseek(fp, set++,0) ==0)

putchar(getc(fp));

fclose(fp);

}

}

}

Перший з трьох аргументів функції fseek () є покажчиком типу FILE на файл, в якому ведеться пошук. Файл слід відкрити, використовуючи функцію fopen (). Другий аргумент "set". Цей аргумент повідомляє, як далеко слід пересунутися від початкової точки (див. Нижче). Він повинен мати значення типу long, яке може бути позитивним (рух вперед) або негативним (рух назад). Третій аргумент є кодом, що визначає початкову точку.

Функція fseek( ) повертає 0, якщо все добре, і -1, якщо є помилка. Оскільки змінна set иніціалізована нулем, при першому проходженні через цикл

while(fseek(fp,set++,0)==0)

putchar(getc(fp));

ми маємо вираз

fseek(fp,OL,0);

що означає, що ми йдемо в файл, на який посилається вказівник fp, і знаходимо байт, віддалений на 0 байт від початку, тобто перший байт. Потім функція putchar () друкує вміст цього байта. При наступному проходженні через цикл змінна set збільшується до 1L, і друкується наступний байт. Тобто, змінна set діє подібно індексу для елементів файлу. Процес триває до тих пір, поки set не спробує потрапити в fseek () після кінця файлу. В цьому випадку fseek () повертає значення -1 і цикл припиняється.

#### Розподіл пам'яті

#### Функція malloc( )

Нехай нам потрібно розподілити пам'ять для запам'ятовування використовуваних даних. Деякі осередки пам'яті розподіляються автоматично, Наприклад, ми можемо оголосити

char str[ ] = "Символьная строка";

Буде виділена *пам'ять*, достатня для запам'ятовування цього рядка. Ми можемо запросити певний обсяг *пам'яті*:

int mas[150];

Це опис виділяє 150 осередків пам'яті, кожна з яких призначена для запам'ятовування цілого значення. Але мова Сі дозволяє нам розподіляти додаткову *пам'ять* під час роботи програми. Припустимо, ми пишемо програму і не знаємо, скільки даних нам доведеться вводити. Тоді можна виділити потрібний нам, за нашим припущенням, обсяг *пам'яті*, а потім, якщо знадобиться, вимагати ще. Щоб зробити це, потрібно використовувати функцію malloc (). І без покажчиків тут не обійтися!

/\* додаємо пам'ять, якщо необхідно \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define STOP "\n" /\* сигнал припинення введення \*/

#define BLOCK 100 /\* байти пам'яті \*/

#define LIM 40 /\* гранична довжина введеного рядка \*/

#define MAX 50 /\* максимальне число вводимих рядків \*/

#define TIME 20000 /\* велика затримка часу \*/

main ( )

{

char store[BLOCK]; /\* вихідний блок пам'яті \*/

char symph[LIM]; /\* приймач вводимих рядків \*/

char \*end; /\* вказує на кінець пам'яті \*/

char \*starts[MAX]; /\* вказує на початок рядків \*/

int index = 0; /\* кількість вводимих рядків \*/

int count; /\* лічильник \*/

char malloc( ); /\* розподільник пам'яті \*/

starts[0]=store;

end=starts[0]+BLOCK-1;

puts("Вводіть рядки по одній ");

puts("для завершення введення на початку рядка натисни клавішу [введення]");

puts("починайте!.");

while(index<MAX) {

if(fgets(symph,LIM,stdin))

if (strcmp(fgets(symph,LIM,stdin),STOP) == 0)

break;

if(strlen(symph)>end - starts[index]) {

/\* дії при нестачі пам'яті для

запам'ятовування даних, що вводяться \*/

puts("почекайте, програма спробує знайти додаткову пам'ять ");

starts[index]=malloc(BLOCK);

end=starts[index]+BLOCK- 1;

for(count=0; count<TIME; count++);

puts("пам'ять знайдена");

}

strcpy(starts[index],symph);

starts[index+1]=starts[index] + strlen(symph)+1;

if(++index<MAX)

printf("Строка: %d. продовжуйте.\n", index);

}

puts("висновок програми ");

for(count=0; count<index; count++)

puts(starts[count]);

}

Давайте подивимося, що робить функція malloc (). Вона бере аргумент у вигляді цілого без знака, яке представляє кількість необхідних байтів пам'яті. Так, malloc (BLOCK) вимагає 100 байт. Функція повертає покажчик на тип char в початок нового блоку пам'яті. Ми використовували опис

char \*malloc( );

щоб попередити компілятор, що malloc () повертає покажчик на тип char. Тому ми присвоїли значення цього покажчика елементу масиву starts [index] за допомогою оператора

starts[index]=malloc(BLOCK);

Припустимо, що ми хочемо працювати з пам'яттю типу int, а не char. Mожете і тут використовувати malloc (). Ось як це робиться:

char malloc( );

/\* як і раніше описуємо як покажчик на char \*/

int \*newmem; newmem = (int \*)malloc(100);

/\* використовуємо операцію приведення типу \*/

Знову потрібно 100 байт. Операція приведення типу перетворює значення, повернене покажчиком на тип char, в покажчик на тип int. Якщо в системі int. займає два байти *пам'яті*, це означає, що 100 байт можна використовувати для запам'ятовування 50 цілих чисел.

#### Функція calloc( )

Іншу можливість розподілу пам'яті дає нам застосування функції calloc( ).

char \*calloc( ); long \*newmem;

newmem=(long \*) calloc(100,sizeof(long));

Функція calloc( ) повертає *покажчик* на char. Потрібно використовувати оператор приведення типу, якщо ви хочете запам'ятати інший тип. calloc( ) має два аргументи, і обидва вони повинні бути цілими без знака. Перший аргумент містить кількість необхідних елементів пам'яті. Другий аргумент - розмір кожного осередку в байтах. Функція calloc( ) обнуляє вміст всього блоку. Ваша *бібліотека* мови Сі можливо представляє кілька інших функцій управління пам'яттю, ви можете досліджувати їх самостійно!