Лекція 2

Масиви та вказівники. Символи та рядки

Масиви в Сі

Масив – впорядкований набір фіксованої кількості однотипних елементів, що зберігаються в послідовно розташованих комірках оперативної пам'яті, мають порядковий номер і спільне ім'я, що надає користувач. Масиви в програмуванні подібні до векторів або матриць в математиці.

Якщо простіше, то масив це сукупність однотипних змінних з одним іменем. Він дозволяє зберігати десятки і сотні елементів під одним іменем.

Властивості масиву в Сі:

- всі елементи мають одне ім’я і один тип даних;
- кожен масив має фіксований розмір – кількість елементів у масиві.
- кожен елемент масиву має власний індекс, по цьому індексу і відбувається звернення до елементів;
- нумерація індексів починається з 0.

Створення масиву

Загальний синтаксис створення масиву:

<тип масиву> ім’я [розмір];

- **<тип масиву>**- будь- який з типів (наприклад: int, bool, char, double) - визначає елементи якого типу, що будуть зберігатися у масиві.
- **ім’я** – аналогічно до змінних та назв функцій – будь-який ідинтифікатор. По цьому імені будуть створюватися звернення до окремих елементів.
- **розмір** – натуральне число, що вказує кількість елементів масиву

Приклад:

```
int a [100];
bool apple[1000];
```

При цьому:

- кількість елементів завжди є натуральним числом або константною змінною;
- замість кількості елементів не можна писати змінні (хоча деякі компілятори це дозволяють).

Таким чином, вираз

```
int a[10];
```

оголошує масив, названий a, що складається з десяти елементів, кожен з типу int. Простіше кажучи, масив є змінною, яка може містити більше одного значення і для того, щоб вказати значення на які треба звернутись в потрібно використовувати числовий індекс.

Масив можна представити наступним чином:

У Сі масиви починаються рахувати з нуля: десять елементів 10-елементного масиву нумеруються від 0 до 9. Нижній індекс, який вказує єдиний елемент масиву - це просто ціле число у квадратних дужках. Першим елементом масиву є a[0], другий елемент - a[1] і т.д. Можна використовувати ці вирази індексів масиву у будь-якому місці, де можна використовувати ім'я простої змінної, наприклад:

```
a[0] = 10;
a[1] = 20;
a[2] = a[0] + a[1];
```

Зауважте, що посилання на індексні масиви (тобто вирази, такі як a[0] і a[1]) можуть з'являтися на будь-якій стороні оператора присвоєння. Індекс не повинен бути константою, як 0 або 1; це може бути будь-який інтегральний вираз. наприклад, звичайний цикл над усіма елементами масиву:

Приклад. Цей цикл встановлює всі десять елементів масиву a до 0.

```
int i;
for(i = 0; i < 10; i++)
a[i] = 0;
```

Декларація масиву

Таким чином, наступні варіанти декларації масиву допустимі в сучасному Сі(Сі++):

1) Декларація масиву сталої довжини:

Тип <ім'я_масиву> [розмір];

Приклад: double ar2[5];

2) Використання довжини ініціалізованою макросом #define:

#define РОЗМІР розмір

Тип <ім'я_масиву> [РОЗМІР];

Приклад:

```
#define NMAX 25
int ar2[NMAX];
```

3) Використання довжини ініціалізованою за допомогою константної натуральної(або цілої) змінної:

const <цілий тип> <РОЗМІР> = розмір;

Тип <ім'я масиву> [РОЗМІР];

Приклад:

```
const size_t N = 10;
unsigned ar3[N];
```

4) (Хак) Ще одним варіантом є визначення константи кількіості елементів масиву за допомогою перерахування (enum):

enum { **<РОЗМІР> = розмір** };
Тип <ім'я масиву> [РОЗМІР];
Приклад:
```
enum{ N = 10};
unsigned ar4[N];
```

5) Допустима також форма створення безрозмірного масиву вигляду:
Тип <ім'я масиву> [];
Приклад:
```
char ar5[];
```

Але якщо ми далі без деяких попередніх дій будемо звертатись до нього, наприклад, ar4[0] = 'A'; то програма може аварійно завершити роботу (що робити для того, щоб цього не було ми розглянемо трохи пізніше) .

Обмеження масивів на Сі

Масиви - це зручне рішення для багатьох проблем, але багато чого не зробить Сі з ними автоматично (так як наприклад, numpy в мові Python). Зокрема, не можна одночасно встановлювати всі елементи масиву і не призначати один масив іншому; обидва присвоєння

```
int a[10];
a = 0;                /* Помилка */
```

та

```
int b[10];
b = a;              /* Помилка */
```

некоректні (хоча другий приклад може зкомпілюватись).

Оскільки замість кількості елементів не можна писати змінні (хоча деякі компілятори це дозволяють), то такий код є також не зовсім правильним для всіх платформ:

```
#include <stdio.h>
int main (){
int n=10;
float arr[n];
}
```

Але це правило не поширюється на константи, через те, що значення константи є незмінним. Цілком можливо таке, що розмір ви будете вказувати декілька разів (заповнення масиву, обробка та інші дії) та може знадобитися замінити всі розміри масиву. Якщо всюди вказувати його у вигляді числа, тоді при заміні ви будете змушені замінювати його всюди де він був прописаний, але якщо ви використовували константу, тоді, щоб замінити всі ці розміри достатньо переписати один рядок коду.

Такий код правильний:

Приклад.

```
#include <stdio.h>
int main ()
{
const int size = 1000;// тип int
double arr[size]; // тип double
return 0;
}
```

Зауважимо, що з точки зору сучасного програмування для розміру масиву краще брати константну натуральну змінну типу size_t, що створено спеціально для зберігання розмірів масивів та рядків (цей тип визначений в stddef.h, але визначиться й при підключенні стандартного stdio.h чи iostream.h в Ci++).

Приклад

```
#include <stdio.h>
int main ()
{
const size_t size = 1000;// тип компілятор визначить оптимальним чином (більшість сучасних платформ визначить long unsigned)
double arr[size]; // масив типу double
return 0;
}
```

Крім цього, варто зазначити, що компілятори Ci та Ci++ не перевіряють вашу програму на таку помилку, як вихід за межі масиву.

Приклад (*поганий*):

```
int a[6]; printf(a[6]);
```

Ця програма скомпілюється але завершиться помилкою при виконанні, оскільки у масиві є 6 елементів, то нумерація буде від [0] до [5], а у наступній команді (printf) ми звертаємося до елемента з індексом [6], якого немає. такі помилки називаються, помилками на етапі виконання (runtime error).

Заповнення даних: ініціалізація масивів

Основне завдання масивів – зберігання даних, тому часто потрібно відразу їх заповнити - ініціалізувати. Ініціалізувати масиви можна декілька способами.

1 спосіб: заповнення масиву зразу ж після його створення.

Синтаксис такий:

тип_даних ім’я_масиву[к-сть_елементів] = {значення_1, значення_2, … , значення_к-сть_ел-тів};}

Приклад:

```
char str[5] = {‘1’, ‘Q’, ‘!’, ‘$’, ‘@’};
```

Крім цього, при такій ініціалізації дозволяється не вказувати кількість елементів (безрозмірний масив).

```
int abr[] = {1, 15, 876};
```

Якщо ви допустите помилку, як у цьому прикладі

```
double abr[5] = {1};
```

тобто розмір більший за ініціалізовані елементи, тоді наступні елементи після 1 будуть заповнені нулями. Тоді вираз буде мати такий вигляд:

```
double abr[5] = {1,0,0,0,0};
{
int a = 1000;
int b[a];
}
```

У іншому випадку, коли розмір менший за кількість заповнених елементів, то виникне помилка на етапі компіляції.

2 спосіб: заповнення масиву за допомогою циклу.

В даному випадку заповнення буде відбуватися з клавіатури. Приклад відповідної програми:

```
#include <stdio.h>
int main ()
{
const size_t size = 25; /* створили константу - розмір. Якщо ми б її не створили, то в кожному з циклів ми були б змушені писати 25, що не лише непорібно але й погано з точки зору зрозумілості та гнучкості коду. У даному випадку для зміни розміру масивів потрібно переписати лише один рядок коду .*/
double arr[size] = {5}; // Далі ми заповнили масив за допомогою першого способу
for (size_t i=0; i<size; ++i){
printf("arr[%lu]=%lf \n",i,arr[i]);
} /* елементи масиву наступні:
5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 (24 нулі). */
for (size_t i=0; i<size; ++i){
  scanf(&arr[i]); // далі за допомогою циклу ми виводимо всі елементи масиву
}
/* Таким чином, елементи можна виводити на екран. по суті на даному моменті ми можемо вважати елементи масиву як змінні, а змінні можна виводити. */
for (int i=0; i<size; ++i){
printf("arr[%lu]=%lf,\t" ,i,arr[i]);
}
/* ми вводимо значення елементів у циклі. також варто замітити, що ми вводимо і елемент з індексом 0, який до цього був рівний 5. тобто значення елементів можна змінювати.*/
getchar();
}
```

Розташування масивів у памяті та багатоіндексні масиви

Найменшою одиницею пам’яті є біт, але при роботі з адресами в комп’ютері, найменшою одиницею вважається байт. В кожному 1 байті міститься 8 біт. Елементи масиву у пам’яті розташовуються у порядку їх зростання, елемент за елементом. крім цього кожний елемент має власний розмір, який дорівнює типу масиву. Якщо масив типу int з кількістю елементів N, то кожен елемент буде займати 4 байти. а розмір всього масиву буде дорівнювати 4*N.

Насправді типом масиву може бути не лише базові типи, а й будь-які типи, що може створити користувач. Зокрема, можна створти масив, кожний з яких має тип x, де x - будь-який тип, наприклад, можна створити масив, кожним з елементів якого є інший масив. Ми можемо використовувати ці масиви масивів для тих самих видів завдань, як ми використовуємо багатовимірні масиви в інших комп'ютерних мовах (або матрицях з математики). Природно, ми не обмежуємося масивами масивів; ми могли б мати масив масивів масивів, які діяли б як тривимірний масив і т.д.

Саме таким чином мова програмування Сі дозволяє створювати та працювати з багатовимірними масивамив. Ось загальна форма оголошення багатовимірного масиву:

<ім'я типу> <імя масиву> [size1][size2]...[sizeN];

Наприклад, наступне оголошення створює тривимірний масив цілих чисел:

int threedim [5][10][4];

Такі декларації потрібно читати "навиворіт". Тобто a2 - це масив з 5 чогось, і що кожен цей чогось є масивом з 7 int. Більш коротко, a2 - це масив розміру 5 з масивів 7-ми int, або a2 - це масив масиву int. Тобто це масив з 5 масивів розміром 7, а не навпаки. Можна також інтерпретувати, що a2 має 5 рядків і 7 стовпців, хоча ця інтерпретація не є обов'язковою. Також можна розглядати "перший" або внутрішній індекс як "x", а другий як "y". Доступ до масиву збігається з тим, що використовуєтсья згідно цієї логіки, як у прикладах нижче.

Двовимірні масиви

Найпростішою формою багатовимірного масиву є двовимірний масив. Двовимірний масив, по суті, є списком одновимірних масивів. Щоб оголосити двовимірний цілочисельний масив розміром [x][y], ви повинні написати щось наступне -

type arrayName [x] [y];

де тип type може бути будь-яким дійсним типом даних C і arrayName буде дійсним C ідентифікатором. Двовимірний масив можна розглядати як таблицю, яка матиме x кількість рядків і y кількість стовпців. Двовимірний масив a, який містить три рядки і чотири стовпці, може бути показаний наступним чином -

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Таким чином, кожен елемент у масиві a ідентифікується назвою елемента форми a [i] [j], де 'a' є назвою масиву, a 'i' і 'j' є індексами, які однозначно ідентифікують. кожен елемент у "a".

Ініціалізація двовимірних масивів

Багатовимірні масиви можуть бути ініціалізовані шляхом вказівки значень з дужками для кожного рядка. Нижче наведено масив з 3 рядками і кожен рядок має 4 колонки.

```
int a[3][4] = {
{0, 1, 2, 3} ,   /*  рядок з індексом 0 */
{4, 5, 6, 7} ,   /*  рядок з індексом 1 */
{8, 9, 10, 11}   /*  рядок з індексом 2 */
};
```

Вкладені фігурні дужки, які вказують потрібний рядок, є необов'язковими. Наступна ініціалізація еквівалентна попередньому прикладу

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Доступ до елемента в двовимірному масиві здійснюється за допомогою індексів, тобто індексу рядків і індексу стовпців масиву. Наприклад

```
int val = a[2][3];
```

Вищенаведена інструкця візьме 4-й елемент з 3-го рядка масиву. Ви можете перевірити його значення на малюнку вище. Давайте перевіримо наступну програму, де ми використовували вкладений цикл для обробки двовимірного масиву

```
#include <stdio.h>

int main () {   /* an array with 5 rows and 2 columns*/
int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};   int i, j;    /* output each array element's value */
for ( i = 0; i < 5; i++ ) {
for ( j = 0; j < 2; j++ ) {
printf("a[%d][%d] = %d\n", i,j, a[i][j] );      }
}
return 0;
}
```

Цей код виведе наступний результат:

```
a[0][0]: 0
a[0][1]: 0
```

```
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

Щоб проілюструвати використання багатовимірних масивів, ми можемо заповнити елементи вищезгаданого масиву a2 за допомогою цього фрагмента коду:

```
int i, j;
for(i = 0; i < 5; i = i + 1)
{
for(j = 0; j < 7; j = j + 1)
a2[i][j] = 10 * i + j;
}
```

Ця пара вкладених циклів встановлює [1] [2] - 12, [4] [1] - 41 і т.д. оскільки перший розмір a2 дорівнює 5, перша індексна змінна індексу, i, виконується від 0 до 4 аналогічно, другий індекс змінюється від 0 до 6. ми можемо надрукувати a2 out (двовимірним способом, пропонуючи його структуру) з подібною парою вкладених циклів:

```
for(i = 0; i < 5; i = i + 1)
{
for(j = 0; j < 7; j = j + 1)
printf("%d\t", a2[i][j]);
printf("\n");
}
```

Друк цих елементів:

```
for(j = 0; j < 7; j = j + 1)
printf("\t%d:", j);
printf("\n");

for(i = 0; i < 5; i = i + 1)
{
printf("%d:", i);
for(j = 0; j < 7; j = j + 1)
printf("\t%d", a2[i][j]);
printf("\n");
}
```

Результат:

```
0:      1:      2:      3:      4:      5:      6:
0:      0       1       2       3       4       5       6
```

1:	10	11	12	13	14	15	16
2:	20	21	22	23	24	25	26
3:	30	31	32	33	34	35	36
4:	40	41	42	43	44	45	46

Як пояснювалося вище, ви можете мати масиви з будь-якою кількістю розмірів, хоча ймовірно, що більшість масивів, які ви створюєте, будуть мати розміри один або два.

Вказівники

В C та C++ існує надзвичайно потужний інструмент для роботи зі складними агрегатами даних, який надає загальний підхід до різних на перший погляд програмних об’єктів таких як масив та рядок. Цей інструмент базується на широкому використанні вказівника.

Поняття вказівника

Кожна змінна у програмі - це об’єкт, який володіє ім’ям і значенням. Після визначення змінної з ініціалізацією всі звернення у програмі до неї за іменем замінюються компілятором на адресу іменованої області оперативної пам’яті, в якій зберігається значення змінної (Рис. 1). Програміст може визначити власні змінні для збереження *адрес областей пам’яті.* Такі змінні називають **вказівниками**.

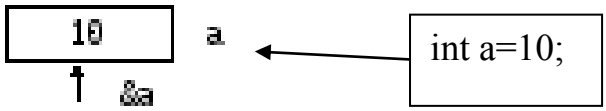


Рис. 1.

Нагадаємо, що кожна змінна в програмі це об'єкт, що має ім'я і значення до якого можна звернутися по імені змінної й отримати значення. Ця змінна в свою чергу десь міститься в пам’яті комп’ютера і адреса пам’яті цієї змінної теж має якесь значення (зазвичай довге беззнакове ціле). Вказівник - це змінна, значення якої є адресою іншої змінної, тобто прямого адреси розташування пам'яті. Іншими словами, вказівник - це символічне представлення адреси змінної в памяті. Вказівник використовується для непрямої адресації змінних і об'єктів.

В C та C++ також існують і змінні типу вказівник(інколи звуть покажчиком). Значенням змінної типу вказівник є адреса змінної або об'єкта. Як і будь-яка змінна або константа, потрібно оголосити вказівник, перш ніж використовувати його для зберігання будь-якої адреси змінної. Загальна форма декларації змінної вказівника:

<тип> *< ідентифікатор>;

або

<тип> *< ідентифікатор> = <початкове значення>;

Ще один із способів завдання вказівника є надання безрозмірного масиву вигляду:

<тип> < ідентифікатор> [ ];

Тут **<тип>** є базовим типом вказівника; він повинен бути типом даних С, а **ідентифікатор** - ім'я змінної покажчика. Зірочка *, що використовується для оголошення вказівника, є тією ж зірочкою, яка використовується для множення. Однак у цьому контексті зірочка використовується для позначення змінної як вказівника. Ось деякі правильні декларації вказівника

```
int    *ip;    /* вказівник на ціле */
double *dp;    /* вказівник на подвійне дійсне число */
float*   fp;    /* вказівник на одинарне дійсне число */
char  *    ch     /* вказівник на символ */
```

Фактичний тип даних значення всіх вказівників, будь то цілочисельний, дійсний, символьний або інший, є однаковим - це довге шістнадцяткове число, яке представляє адресу пам'яті. Єдиною відмінністю між вказівниками різних типів даних є тип даних змінної або константи, на яку вказує вказівник.

Приклад:

char ch;	Тут ch- значення символьного типу,
char *cptr;	cptr - вказівник на значення символьного типу,
int val,*ivptr, n;	val і n - значення цілого типу, а *ivptr - вказівник на значення цілого типу;
double r,*rp;	r- змінна дійсного типу подвійної точності, а *rp - вказівник на змінну такого ж типу.

Операції над вказівниками

Нехай змінна типу вказівник має ім'я ptr, тоді в якості значення їй можна присвоїти адресу за допомогою наступного оператора:

```
ptr=&vr;
```

В мовах С та С++ при роботі з вказівниками велике значення має операція непрямої адресації (розіменування) *. Операція * дозволяє звертатися до змінної не напряму, а через вказівник, який містить адресу цієї змінної. Ця операція є унарною та має асоціативність зліва направо. Цю операцію не слід плутати з бінарною операцією множення.

Виведення значення вказівника можна робити як виведення рядка або цілого числа, але за стандартом це робиться за допомогою специфікатора "%p":

```
printf("Address  %p\n", ptr );
```

Існують наступні операції з вказівниками:

- **декларація змінної вказівника;**

- **присвоєння значення або ініціалізація;**
- **присвоєння адреси змінній вказівника (оператор &);**
- **отримання значення (розіменування) за адресою, доступною в вказівнику;**
- **інкремент вказівника;**
- **декремент вказівника;**
- **додавання цілого числа до вказівника;**
- **віднімання цілого числа від вказівника;**
- **порівняння вказівників.**

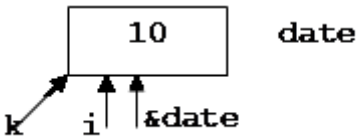
Декларація змінної вказівника:

Вказівник - це змінна, значенням якої служить адреса об'єкта конкретного типу. Щоб визначити вказівник треба повідомити на об'єкт якого типу посилається цей вказівник.

```
char *z;
int *k,*i;
float *f;
```

Оператор присвоювання (=) виконує зворотну дію: імені змінної ставиться у відповідність значення.

Приклад:




```
int date = 10;
int *i, *k;;
i = &date;
k = i;
z = NULL;
```

Присвоєння адреси змінній вказівника

Також є **операція визначення адреси — &**, за допомогою якої визначається адреса комірки пам’яті, що містить задану змінну. наприклад, якщо vr — ім’я змінної, то &vr — адреса цієї змінної.

Операція розіменування (*). Операндом цієї операції завжди є вказівник. Результат операції - це той об'єкт, що адресує вказівник_операнд.

Нехай ptr — вказівник, тоді *ptr — це значення змінної, на яку вказує ptr.

Наступний приклад використовує ці операції -

```
#include <stdio.h>
int main () {
int  var = 20;   /* ініціалізація цілої змінної var */
int  *ip;       /* декларація вказівника на ціле число*/
ip = &var;  /* зберігаємо у вказівнику адресу var */
//printf("Address of var variable: %x\n", &var  можна виводити як ціле %x
але буде попередження компілятора  (not int*)
printf("Address of var variable: %p\n", &var  ); //специфіктор адреси %p
/* виводимо значення вказівника */
```

```
printf("Address stored in ip variable: %p\n", ip );
/* виводимо значення цілої змінної за адресом вказівника */
printf("Value of *ip variable: %d\n", *ip )
}
```

Результат роботи коду:

```
Address of var variable: 0x7ffc085bfe18
Address stored in ip variable: 0x7ffc085bfe18
Value of *ip variable: 20
```

Вказівник на NULL

Завжди є гарною практикою присвоювати значення змінній NULL у випадку, якщо ви не маєте точної адреси. Це робиться під час оголошення змінної. Покажчик, призначений NULL, називається нульовим покажчиком.

В мові Сі нульова адреса позначається константою NULL, що визначена в заголовному файлі stdio.h. В сучасному Сі++ радять користуватися nullpointer.

Покажчик NULL - це константа з значенням нуля, визначеним у декількох стандартних бібліотеках. Розгляньте наступну програму -

```
#include <stdio.h>
int main () {
int  *ptr = NULL;
printf("The value of ptr is : %x\n", ptr  );
return 0;
}
```

Результат:

```
The value of ptr is 0
```

У більшості операційних систем програми не мають доступу до пам'яті на адресу 0, оскільки ця пам'ять зарезервована операційною системою. Однак адреса пам'яті 0 має особливе значення; він сигналізує, що вказівник не призначений для вказівки на доступну пам'ять. Але за умовами, якщо покажчик містить нульове (нульове) значення, передбачається, що він вказує на нічого.

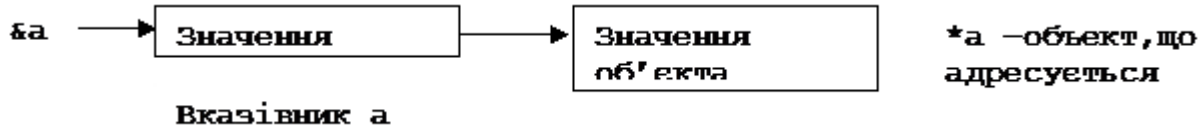
Щоб перевірити нульовий покажчик, ви можете використовувати оператор "if" наступним чином -

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

Арифметичні операції над вказівниками

Вказівник у Сі - адреса, що є числовим значенням. Отже, ви можете виконувати арифметичні операції на вказівнику так само, як і на числовому значенні. Є чотири арифметичних оператори, які можна використовувати на покажчиках: інкремент(++), декремент(--) та додавання і віднімання цілого числа.

Подібно будь-яким змінної змінна типу вказівник має ім'я, адрес у пам'яті і значення.



За допомогою унарних операцій інкремент(++) і декремент(–-) числові значення змінних типу вказівник міняються по різному, у залежності від типу даних, з яким зв'язані ці змінні.

Приклад:

```
char *z;
int *k,*i;
float *f;
z++; // значення змінюється на 1
i++; // значення змінюється на 2
f++; // значення змінюється на 4
```

Тобто при зміні вказівника на 1, вказівник переходить до початку наступного (попереднього) поля тієї довжини, що визначається типом об'єкта, який адресується вказівником.

Щоб зрозуміти арифметику вказівника, припустимо, що ptr є цілим числом вказівником, що вказує на адресу 1000. Якщо взяти 32-бітні цілі числа, зробимо наступну арифметичну операцію на вказівнику: ptr ++

Після зазначеної вище операції, ptr буде вказувати на розташування 1004, оскільки кожен раз ptr збільшується, він вказує на наступне ціле розташування, яке становить 4 байти поряд (пам’ятаємо, що ціле число – це скоріше за все 4 байти) з поточним місцем розташування. Ця операція перемістить вказівник на наступну пам'ять, не впливаючи на фактичне значення в пам'яті. Якщо ptr вказує на символ, адреса якого є 1000, то вищезгадана операція вкаже на розташування 1001, оскільки наступний символ буде доступний на 1001.

Інкремент та збільшення вказівника

Інколи зручно використовувати вказівник у нашій програмі замість масиву, оскільки вказівник змінної може бути збільшений, на відміну від імені масиву, який не може бути збільшений, оскільки він є постійним вказівником. Наступна програма збільшує вказівник змінної для доступу до кожного наступного елементу масиву

```
#include <stdio.h>
const int MAX = 3;
int main () {
int  var[] = {10, 100, 200};
int  i, *ptr;
/* встановимо address вказіника (pointer) */
ptr = var;
```

```
for ( i = 0; i < MAX; i++) {
printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
/* рухаємося далі по масиву */
ptr++;
}
return 0;
}
```

Результат роботи коду–

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

Крім того, вказівник дозволяє додавати не лише одиницю за допомогою інкременту але й будь яке ціле число подобним чином (за допомогою **оператору += або просто +):**

```
int  var[] = {10, 100, 200};
int  i, *ptr;
printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
/* move to the next location */
ptr+=2;
printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
```

Декремент та віднімання цілого числа від вказівника

Так само як інкремент, можна робити й операцію декременту над вказівниками:

```
#include <stdio.h>
const int MAX = 3;
int main () {
int  var[] = {10, 100, 200};
int  i, *ptr;
/* візьмемо адресу останнього елементу масиву*/
ptr = &var[MAX-1];

for ( i = MAX; i > 0; i--) {
   printf("Address of var[%d] = %x\n", i-1, ptr );
```

```
   printf("Value of var[%d] = %d\n", i-1, *ptr );
   /* рухаємося назад по масиву */
   ptr--;
}
return 0;
}
```

Результат:

```
Address of var[2] = bfedbcd8
Value of var[2] = 200
Address of var[1] = bfedbcd4
Value of var[1] = 100
Address of var[0] = bfedbcd0
Value of var[0] = 10
```

Аналогічно, можна віднімати будь яке ціле число подобним чином (за допомогою **оператору -= або просто -):**

Порівняння вказівників

Вказівники можна порівняти за допомогою реляційних операторів, таких як ==, <, та >. Якщо p1 і p2 вказують на змінні, пов'язані один з одним, такі як елементи одного і того ж масиву, то p1 і p2 можуть бути змістовно порівняні.

Наступна програма змінює попередній приклад - один, збільшуючи покажчик змінної, доки адреса, на яку він вказує, є меншим або рівним адресою останнього елемента масиву, тобто &var[MAX-1] :

```
#include <stdio.h>
const int MAX = 3;
int main () {
int  var[] = {10, 100, 200};
int  i, *ptr;
/* візьмемо адресу першого елементу масиву */
ptr = var;
i = 0;
while ( ptr <= &var[MAX - 1] ) {
   printf("Address of var[%d] = %x\n", i, ptr );
   printf("Value of var[%d] = %d\n", i, *ptr );
   ptr++;
   i++;
}
return 0;
}
```

Результат:

```
Address of var[0] = bfdbcb20
```

```
Value of var[0] = 10
Address of var[1] = bfdbcb24
Value of var[1] = 100
Address of var[2] = bfdbcb28
Value of var[2] = 200
```

Вкладені вказівники та масиви вказівників

Може виникнути ситуація, коли ми хочемо зберегти масив, який може зберігати покажчики на int або char або будь-який інший доступний тип даних. Далі йде оголошення масиву покажчиків на ціле число:

```
int *ptr[MAX];
```

Він оголошує ptr як масив MAX цілих покажчиків. Таким чином, кожен елемент у ptr утримує покажчик на значення int. У наступному прикладі використовуються три цілих числа, які зберігаються в масиві покажчиків:

```
#include <stdio.h>
static const int MAX = 3;

int main () {
   int  var[] = {10, 100, 200};
   int i, *ptr[MAX];
   for ( i = 0; i < MAX; i++) {
            ptr[i] = &var[i]; /* взяти адресу integer. */
}

for ( i = 0; i < MAX; i++) {
   printf("Value of var[%d] = %d\n", i, *ptr[i] );
}
return 0;
}
```

Результат роботи:

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

Використання вказівників при роботі з масивами

Ім'я масиву без індексу є вказівником-константою, тобто адресою першого елемента масиву (a[0]).

```
↓       a
        *a = = a[0] ;
*(a+1) = = a[1];
. . . . . .
*(a+i) = =a[i];
```

Відповідно до синтаксису в С існують тільки одномірні масиви, але їх елементами , у свою чергу, теж можуть бути масиви.

int a[5][5];

Для двовимірного масиву:

a[m][n] = = *(a[m]+n) = = *(*(a+m)+n);

Приклад1. Опис вказівників.

int *ptri; //вказівник на змінну цілого типу

char *ptrc; //вказівник на змінну символьного типу

float *ptrf; //вказівник на змінну з плаваючою точкою

Такий спосіб оголошення вказівників виник внаслідок того, що змінні різних типів займають різну кількість комірок пам'яті. При цьому для деяких операцій з вказівниками необхідно знати об'єм відведеної пам'яті. Операція * в деякому розумінні є оберненою до операції &.

Вказівники використовуються для роботи з масивами. розглянемо оголошення двовимірного масиву:

int mas[4][2];

int *ptr;

Тоді вираз ptr=mas вказує на першу колонку першого рядка матриці. Записи mas і &mas[0][0] рівносильні. Вираз ptr+1 вказує на mas[0][1], далі йдуть елементи: mas[1][0], mas[1][1], mas[2][0] і т. д.; ptr+5 вказує на mas[2][1].

Двовимірні масиви розташовані в пам’яті так само, як і одновимірні масиви, займаючи послідовні комірки пам’яті

ptr	ptr+1	ptr+2	ptr+3	ptr+4	ptr+5
mas[0][0]	mas[0][1]	mas[1][0]	mas[1][1]	mas[2][0]	mas[2][1]

Масив, розмірність якого стає відомою в процесі виконання програми, називається д**инамічним.**

Робота з пам’яттю в С

В попередніх програмах ми використовували вказівника на вже існуючі змінні. Інколи потрібно передавати у вказівники адреси змінних, що ми повинні визначити по ходу виконання програми. Для цього потрібно виділити місце в пам’яті, де будуть записуватися та зберігатися до кінця програми значення цих змінних.

Розглянемо роботу пам’яті комп’ютера при виконанні роботи програми.

Види пам’яті

Відкомпільована С-програма створює та використовує 4 логічно відокремлені ділянки пам’яті, що мають власні призначення.

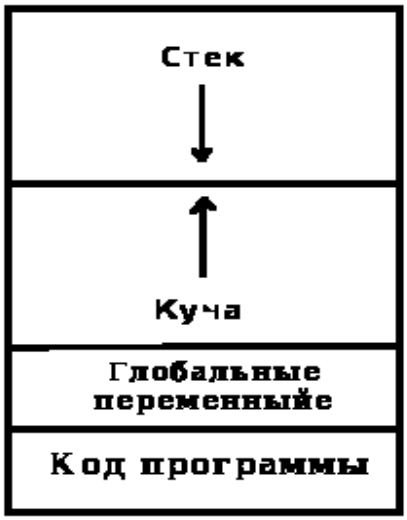
Перша ділянка - це пам’ять, що містить код програми.

Наступна ділянка призначена для зберігання глобальних змінних.

Стек - швидко доступна ділянка пам’яті, що може використовуватись для різноманітних цілей при виконанні програми. Він містить адреси повернень функцій, що викликаються, аргументи, що передаються в функцію та локальні змінні. Він також використовується для зберігання поточного стану процесору.

Куча - це ділянка вільної пам’яті, яку програма може використовувати для динамічного виділення пам’яті під різноманітні програмні об’єкти великого розміру, зокрема класи, структури, масиви й таке інше.

Концептуальна карта пам’яті що виділяє програма на С.



Запуск програм C / C ++ зазвичай складається з таких дій, які виконуються в описаному порядку:

1. Вимкнути всі переривання.
2. Копіювати будь-які ініціалізовані дані з ПЗУ в ОЗУ.
3. Занулити неініціалізовану область даних.
4. Виділити простір і ініціалізувати стек.
5. Ініціалізувати покажчик стека процесора.
6. Створіть і ініціалізувати кучу.
7. Виконати конструктори та ініціалізатори для всіх глобальних змінних (тільки C ++).
8. Увімкнути переривання.
9. Виклик основної функції.

Як правило, код запуску також буде включати кілька інструкцій після виклику main. Ці інструкції будуть виконуватися тільки в тому випадку, якщо виконується програма мови високого рівня (тобто, виклик основних функцій повертається). Залежно від характеру вбудованої системи, ви можете використовувати ці інструкції для припинення роботи процесора, скидання всієї системи або передачі керування засобом налагодження.

Оскільки код запуску не вставляється автоматично, програміст повинен, як правило, сам збирати його і включати отриманий об'єктний файл зі списку вхідних файлів до лінкера. Йому навіть може знадобитися спеціальний параметр командного рядка, щоб змусити його не вставляти звичайний код запуску. Робочий код запуску для різних цільових процесорів можна знайти у пакеті GNU під назвою libgloss.

Кілька потоків живуть в одному процесі в одному просторі, кожен потік виконуватиме конкретне завдання, мати свій власний код, власну пам'ять стека, покажчик інструкцій і спільну пам'ять. Якщо потік має витік пам'яті, він може пошкодити інші потоки і батьківський процес.

Сам вказівник є змінною, що виділяється в програмному стеку. Сам він означає адресу змінної що виділяє пам’ять «на кучі».

Для того, щоб працювати з динамічним масивом потрібно виділити відповідну пам’ять під цей масив. Тобто вказати компілятору, щоб він зарезервував місце у відповідній ділянці пам’яті для того, щоб записувати туди дані. Крім того це потрібно щоб інші змінні, сам компілятор та інші програми не займали це місце. Якщо цього не зробити то будуть серйозні помилки, які можуть призвести до зупинки не лише програми але й всієї системи. Крім того, ще однією помилкою є звільнення зарезервованої ділянки пам’яті після виконання програми або потрібної функції. На жаль компілятор Сі(Сі++) не звільнить виділену пам’ять автоматично після виходу функції або програми і таким чином залишиться зайнятою ділянка пам’яті. Це так званий витік пам’яті (Memory leak). Він може призвести до того, що наступні ділянки програми або повторні виклики чи багатопоточні виклики цієї програми будуть «падати» з невідомих причин або вішати систему. На жаль зроблена така помилка на буде повідомлена компілятором, а подальше падіння програми буде виникати не в тому місці де була зроблена помилка, що ускладнює процес відлагодження програми. Для боротьби з такими помилками потрібні спеціальні інструменти та спеціальні навички дебага, назразок використання програми Valgrind.

Виділення пам’яті

Для виділення пам’яті вказівнику можна використовувати такі стандартні функції:

void* malloc(size_t size);
void* calloc(size_t num, size_t size);
void *realloc(void *ptr, size_t new_size);

malloc - виділення пам’яті

void* malloc(size_t size);

Метод визначений в заголовочному файлі <stdlib.h>

Визначає розміри байтів неініціалізованого сховища.

Якщо розподіл успішно завершується, повертає покажчик на найнижчий (перший) байт у виділеному блоці пам'яті, який відповідним чином вирівняний для будь-якого типу об'єкта з фундаментальним вирівнюванням.

Якщо розмір дорівнює нулю, поведінка є визначеною реалізацією (може бути повернуто нульовий покажчик або може бути повернений деякий ненульовий покажчик, який може не використовуватися для доступу до сховища, але повинен бути переданий вільному). malloc є потокобезпечним:

він веде себе так, ніби тільки отримує доступ до пам'яті, видимої через її аргумент, а не будь-яку статичну пам'ять.

Параметри:

- **size** - розмір(натуральне число) - кількість виділених байт

При успішному завершенні повертає вказівник на початок знову виділеної пам'яті. Щоб уникнути витоку пам'яті, повернений покажчик повинен бути звільнений з free () або realloc ().

При відмові повертає нульовий покажчик.

Приклад:

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

int main(void)
{
int *p1 = malloc(4*sizeof(int));  // виділяє пам'ять під масив 4 int
int *p2 = malloc(sizeof(int[4])); // аналогічно,
int *p3 = malloc(4*sizeof *p3);   // що один варіант цього самого

if(p1) { // перевіряємо, чи коректно виділена пам'ять
for(int n=0; n<4; ++n) // проходимо масив та ініціалізуємо його
   p1[n] = n*n;
for(int n=0; n<4; ++n) // друкуємо значення масиву
          printf("p1[%d] == %d\n", n, p1[n]);
}

   free(p1);
   free(p2);
   free(p3);
}
```

Output:

```
p1[0] == 0
p1[1] == 1
p1[2] == 4
p1[3] == 9
```

calloc - ініціалізація виділеної памяті

void* calloc(size_t num, size_t size);

Метод визначений в. <stdlib.h> .

Виділяє пам'ять для масиву num об'єктів розміру розміру і ініціалізує всі байти в виділеному сховищі ***нулями***.

Якщо розподіл успішно завершується, повертає покажчик на найнижчий (перший) байт у виділеному блоці пам'яті, який відповідним чином вирівняний для будь-якого типу об'єкту.

Якщо розмір дорівнює нулю, поведінка визначена реалізацією (може бути повернуто нульовий покажчик або може бути повернений деякий ненульовий покажчик, який може не використовуватися для доступу до сховища)

calloc є потокобезпечним: він веде себе так, ніби тільки отримує доступ до пам'яті, видимої через його аргумент, а не будь-яку статичну пам'ять.

Параметри:

- **num** - кількість обєктів
- **size** - розмір кожного обєкту

При успішному завершенні повертає вказівник на початок знову виділеної пам'яті. Щоб уникнути витоку пам'яті, повернений покажчик повинен бути звільнений з free () або realloc ().

При відмові повертає нульовий покажчик.

Примітки: Завдяки вимогам вирівнювання кількість виділених байт не обов'язково дорівнює num * **size**.

Приклад

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

int main(void)
{
int *p1 = calloc(4, sizeof(int));    // allocate and zero out an array of 4 int
int *p2 = calloc(1, sizeof(int[4])); // same, naming the array type directly
int *p3 = calloc(4, sizeof *p3);     // same, without repeating the type name

if(p2) {
for(int n=0; n<4; ++n) // print the array
        printf("p2[%d] == %d\n", n, p2[n]);
}

   free(p1);
   free(p2);
   free(p3);
}
```

Output:

```
p2[0] == 0
p2[1] == 0
p2[2] == 0
p2[3] == 0
```

realloc - перерозподіл памяті

```
void *realloc( void *ptr, size_t new_size );
```

Метод визанчений в. <stdlib.h>

Перерозподіляє задану область пам'яті. Пам'ять повинна бути попередньо виділена за допомогою malloc (), calloc () або realloc () і ще не звільнена за допомогою виклику free або realloc. В іншому випадку результати не визначені.

Перерозподіл здійснюється або:

а) розширення або стискання існуючої ділянки, на яку вказують ptr, якщо це можливо. Вміст області залишається незмінним до меншої кількості нових і старих розмірів. Якщо область розширена, вміст нової частини масиву не визначено.

б) виділяючи новий блок пам'яті розміру new_size байтів, копіюючи область пам'яті з розміром, рівним меншому з нових і старих розмірів, і звільняючи старий блок.

Якщо пам'яті недостатньо, старий блок пам'яті не звільняється і повертається нульовий покажчик.

Якщо ptr є NULL, поведінка є такою ж, як виклик malloc (new_size). Якщо new_size дорівнює нулю, поведінка визначена реалізацією (може бути повернуто нульовий покажчик (у цьому випадку старий блок пам'яті може бути або не може бути звільнений), або може бути повернений деякий ненульовий покажчик, який може не використовуватися для доступу до сховища) . Метод realloc є потокобезпечним: він веде себе так, ніби тільки отримує доступ до пам'яті, видимої через її аргумент, а не будь-яку статичну пам'ять.

Параметри:

- **ptr** - вказівник на область пам'яті, яку потрібно перерозподілити
- **new_size** - новий розмір масиву в байтах.

При успішному завершенні повертає вказівник на початок знову виділеної пам'яті. Щоб уникнути витоку пам'яті, повернений вказівник повинен бути звільнений з free () або realloc (). Оригінальний ptr покажчика недійсний, і будь-який доступ до нього є невизначеною поведінкою.

При відмові повертає нульовий покажчик. Оригінальний ptr вказівника залишається дійсним і може бути необхідним для звільнення з free () або realloc ().

Приклад

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

int main(void)
{
int *pa = malloc(10 * sizeof *pa); // allocate an array of 10 int
if(pa) {
      printf("%zu bytes allocated. Storing ints: ", 10*sizeof(int));
for(int n = 0; n < 10; ++n)
         printf("%d ", pa[n] = n);
```

```
}

int *pb = realloc(pa, 1000000 * sizeof *pb); // reallocate array to a larger size
if(pb) {
    printf("\n%zu bytes allocated, first 10 ints are: ", 1000000*sizeof(int));
for(int n = 0; n < 10; ++n)
        printf("%d ", pb[n]); // show the array
    free(pb);
} else { // if realloc failed, the original pointer needs to be freed
    free(pa);
}
}
Output:
40 bytes allocated. Storing ints: 0 1 2 3 4 5 6 7 8 9
4000000 bytes allocated, first 10 ints are: 0 1 2 3 4 5 6 7 8 9
```

aligned_alloc

З стандарту C11 в <stdlib.h> існує також функція aligned_alloc:

void *aligned_alloc(size_t alignment, size_t size);

Виділяє байти неініціалізованого сховища, вирівнювання якого задається параметром вирівнювання. Параметр розміру повинен бути цілим та кратним до вирівнювання. alignment_alloc є потокобезпечним: він веде себе так, ніби тільки отримує доступ до пам'яті, видимої через її аргумент, а не будь-яку статичну пам'ять.

Попередній виклик до free або realloc, який звільняє область пам'яті, синхронізується з викликом вирівнювання, який виділяє ту ж саму або частину тієї ж області пам'яті. Ця синхронізація відбувається після будь-якого доступу до пам'яті за допомогою функції, що звільняється, і перед будь-яким доступом до пам'яті вирівнювання. Існує єдиний повний порядок всіх функцій розподілу і вивільнення, що діють на кожній конкретній області пам'яті.

Параметри:

alignment - вказує вирівнювання. Має бути допустиме вирівнювання, що підтримується реалізацією.

size - кількість виділених байт. Ціле, що є кратним вирівнюванню.

При успішному завершенні повертає покажчик на початок знову виділеної пам'яті. Щоб уникнути витоку пам'яті, повернений покажчик повинен бути звільнений з free () або realloc (). При відмові повертає нульовий покажчик.

Примітки: Передача розміру, який не є кратним вирівнюванню або вирівнюванням, що не є дійсним або не підтримується реалізацією, призводить до помилки функції і повернення нульового покажчика.

Звичайний malloc вирівнює пам'ять, придатну для будь-якого типу об'єкта (який на практиці означає, що він вирівнюється до alignof (max_align_t)). Ця функція є корисною для обробки виключнизх ситуацій, наприклад, для SSE, кешу або VM.

Приклад

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

int main(void)
```

```
{
int *p1 = malloc(10*sizeof *p1);
    printf("default-aligned addr:   %p\n", (void*)p1);
    free(p1);

int *p2 = aligned_alloc(1024, 1024*sizeof *p2);
    printf("1024-byte aligned addr: %p\n", (void*)p2);
    free(p2);
}
Possible output:
default-aligned addr:   0x1e40c20
1024-byte aligned addr: 0x1e41000
```

Звільнення памяті

Для звільнення виділеної пам’яті крім realloc використовується команда free:

```
void free( void* ptr );
```

Це є функція визначена в <stdlib.h>.

Звільнює простір, попередньо виділений malloc (), calloc (), alignment_alloc, (починаючи з C11) або realloc ().

Якщо ptr є нульовим покажчиком, функція нічого не робить.

Поведінка є невизначеною, якщо значення ptr не дорівнює значенню, повернутому раніше malloc (), calloc (), realloc (), або alignment_alloc () (починаючи з C11).

Поведінка є невизначеною, якщо область пам'яті, на яку посилається ptr, вже була звільнена, тобто вільна () або realloc () вже викликана з ptr як аргумент, а виклики до malloc (), calloc () або realloc ( ) після чого покажчик дорівнює ptr.

Поведінка не визначена, якщо після повернення free (), доступ здійснюється через покажчик ptr (якщо інша функція виділення не привела до значення покажчика, рівного ptr). free є потокобезпечним: він веде себе так, ніби тільки отримує доступ до пам'яті, видимої через її аргумент, а не будь-яку статичну пам'ять.

Параметри:

- **ptr** - вказівник на пам'ять для звільнення

Поверненого значення немає.

Примітки: функція приймає (і нічого не робить) з нульовим вказівником, щоб зменшити кількість спеціального корпусу. Незалежно від того, чи буде успішне виділення, вказівник, що повертається функцією розподілу, може бути переданий до free ()

Приклад

```
#include <stdlib.h>
int main(void)
{
int *p1 = malloc(10*sizeof *p1);
```

```
free(p1); // все що виділено(allocated) як вказівник повинно бути звільнено
int *p2 = calloc(10, sizeof *p2);
int *p3 = realloc(p2, 1000*sizeof *p3);
if(p3) // якщо p3 не null це значить що p2 звільнено realloc
free(p3);
else // якщо p3 null означає що p2 ще не звільнено
free(p2);
}
```

C++ робота з пам'яттю

В C++ для роботи з динамічними об'єктами використовують спеціальні операції new і delete. За допомогою операції new виділяється пам'ять під динамічний об'єкт (який створюється в процесі виконання програми), а за допомогою операції delete створений об'єкт видаляється з пам'яті.

Приклад . Виділення пам'яті під динамічний масив.

Нехай розмірність динамічного масиву вводиться з клавіатури. Спочатку необхідно виділити пам'ять під цей масив, а потім створений динамічний масив треба видалити.

```
…
unsigned n;
scanf("%u",&n); // n — розмірність масиву
int *mas=new int[n]; // виділення пам'яті під масив
delete [] mas; // звільнення пам'яті
…
```

В цьому прикладі mas є вказівником на масив з n елементів. Оператор int *mas=new int[n] виконує дві дії: оголошується змінна типу вказівник, далі вказівнику надається адреса виділеної області пам'яті у відповідності з заданим типом об'єкта.

Для цього ж прикладу можна задати наступну еквівалентну послідовність операторів:

```
…
int *mas;
unsigned n;
scanf("%u",&n); // n — розмірність масиву
mas=new int[n];// виділення пам'яті під масив
delete [] mas;// звільнення пам'яті
```

Якщо за допомогою операції new неможливо виділити потрібний об'єм пам'яті, то результатом операції new є 0.

Іноді при програмуванні виникає необхідність створення багатовимірних динамічних об'єктів. Програмісти-початківці за аналогією з поданим способом створення одновимірних динамічних масивів для двовимірного динамічного масиву розмірності n*k запишуть наступне

```
mas=new int[n][k]; // Невірно! Помилка!
```

Такий спосіб виділення пам’яті не дасть вірного результату. Наведемо приклад створення двовимірного масиву.

```
#include<iostream.h>
#include<conio.h>
int main()
{
int n;const m=5;
printf("input the number";
scanf(&n);
int** a; //a - вказівник на масив вказівників на рядки
a=new int* [n]; //виділення пам’яті для масиву вказівників на n рядків
for(int i=0;i<n;i++)
a[i]=new int [m]; //виділення пам’яті для кожного рядка масиву розмірністю nxm
…
for(int i=0;i<n;i++)
{for(int j=0;j<m;j++)
printf(a[i][j]);
}
for(int i=0;i<n;i++)
delete [] a[i]; //звільнення пам’яті від кожного рядка
delete [] a; //звільнення пам’яті від масиву вказівників
getch();
return 0;
}
}
```

Рядки (Null-terminated byte strings)

Символьний рядок представляє собою набір з одного або більше символів.

Приклад : "Це рядок"

В мові Сі немає спеціального типу даних, який можна було б використовувати для опису рядків. Замість цього рядки представляються у вигляді масиву елементів типу *char*. Тобто в С для зберігання рядків використовують символьні масиви. Це такі ж масиви як і ті що розглядалися раніше, але зберігають вони не числові дані, а символьні. Це означає, що символи рядка розташовуються в пам'яті в сусідніх комірках, по одному символу в комірці.

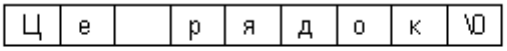


Рис. 1.13. Представлення рядка у вигляді масиву символів

Необхідно відмітити, що останнім елементом масиву є символ '\0'. Це нульовий символ (байт, кожний біт якого рівний нулю). У мові Сі він використовується для того, щоб визначати кінець рядка.

Можна уявити символи такого масиву розташованими послідовно в сусідніх комірках пам'яті - в кожному осередку зберігається один символ і займає один байт. Один байт тому, що кожен елемент символьного масиву має тип char. Останнім символом кожної такого рядка є символ \ 0 (нульовий символ). Наприклад: «Текст: перший рядок \n\r Text \t Кінець. \0»

Сам текст, включаючи пробіл та спецсимволи, складається з певної кількості символів. Якби в останній комірці перебувала наприклад '.' (Крапка), а не нульовий символ '\0' - для компілятора це вже не рядок. І працювати з таким набором символів треба було б, як зі звичайним масивом - записувати дані в кожну клітинку окремо і виводити на екран посимвольно (за допомогою циклу):

Приклад.

```
char str [16] = { 'T', 'e', 'x', 't', ' ', 'a', 'n', 'd', ' ', 's', 't', 'o', 'p', '!' ,'\0'};
for (int i = 0; i <=sizeof(str); i++)
{
   printf("%c",str[i]);
}
printf("\n");
```

На щастя в C ++ є куди більш зручний спосіб ініціалізації і звернення до символьних масивів - один рядок. Для цього останнім символом такого масиву обов'язково повинен бути нульовий символ \ 0. Саме він робить набір символів рядком, працювати з якою, набагато легше, ніж з масивом символів. Цей рядок зветься *символьним рядком* в С, або рядком байтів з нульовим завершенням.

Рядок байтів з нульовим завершенням (NTBS) - це послідовність ненульових байтів, за якими слідує байт з нульовим значенням (символ завершення нуля). Кожен байт у рядку байтів кодує один символ деякого набору символів. Наприклад, масив символів {'x63', 'x61', 'x74', '0'} є NTBS, що містить рядок "cat" в кодуванні ASCII.

***Символьний рядок* (*стрінг*)** — це масив символів, що закінчується у лапки ("). Він має тип char. Нульовий символ (\0) автоматично додається останнім байтом символьного рядка та виконує роль ознаки його кінця. Кількість елементів у масиві дорівнює кількості символів у рядку плюс один, оскільки нульовий символ також є елементом масиву. Кожна рядкова константа, навіть у випадку, коли вона ідентична іншій рядковій константі, зберігається у окремому місці пам'яті. Якщо необхідно ввести у рядок символ лапок ("), то перед ним треба поставити символ зворотного слешу (\). У рядок

можуть бути введені будь-які спеціальні символьні константи, перед якими стоїть символ \.

Основні методи ініціалізації символьних рядків.

• char str1[]= "ABCdef";
• char str2[]={'A', 'B', 'C', 'd', 'e', 'f','\0'};
• char str3[100]; gets(str3);

Цей спосіб вважається зараз небезпечним з точки зору Buffer Overflow, тому краще скористатись наступною формою, де вказується довжина рядку:

fgets(str3, 100, stdin);

При цьому потрібно вводити не більше символів ніж в розмірі буферу (в даному випадку - 100).

• char str4[100];
scanf("%s",str4);

Помітимо, що в цій формі непотрібно використовувати амперсанд перед змінною str4 (подумайте самі, чому). Зауважимо, що ця форма теж може бути Buffer Overflow небезпечною, тому можна використовувати fscanf(stdin,"%s", name);

Помітимо, що при даному способі вводу вводитись буде лише до першого пробілу, що введений в тексті, тому якщо потрібно ввести саме рядок з пробілами, то потрібно використовувати форму scanf(" %[^\n]s",name); (див. таблицю специфікаторів scanf та printf). Подібна стратегія потрібна також і в випадках якщо потрібно обробити інші роздільники.

Декларація рядку

Оголошується рядок таким чином - створюємо масив типу char, розмір в квадратних дужках вказувати не обов'язково (його підрахує компілятор), оператор = і в подвійних лапках пишемо необхідний текст. Тобто ініціализуємо масив рядковою константою:

```
#include <stdio.h>
#include <locale.h>
int main (){
    setlocale (LC_ALL, "Ukrainain");
    char str2[] = "Текст"; // '\0' присутній неявно
    printf("str=%s\n",str2);
    return 0;
}
```

Прописувати нульовий символ не треба. Він присутній неявно і додається в кожну таку строкову константу автоматично. Таким чином, при тому що ми бачимо текст "text" з 4 символів в рядку, розмір масиву буде 5, так як \ 0 теж символ і займає один байт пам'яті. Займе він останній символ цього символьного масиву.

Примітка. Це може не бути правдою для масиву, що записаний не латинським алфавітом, а наприклад кирилицею, бо кириличний символ може займати більше ніж один байт. Наприклад на моєму комп'ютері текст з попереднього прикладу займає 2*5+1 =11 байт, бо українські літери займають 2 байти.

Примітка до примітки. В сучасному С для підтримки різних алфавітів та кодувань (наприклад UTF-8) можна використовувати тип символу wchar_t (мультибайтовий символ).

Як бачите, для виведення рядка на екран, досить звернутися до неї по імені: printf("%s",str); буде виводити на екран символ за символом, поки не зустріне в одній з комірок масиву символ кінця рядка \ 0 і висновок перерветься. Таке звернення для звичайного символьного масиву (масиву без \ 0) не є правильним: Так як компілятор виводив би символи на екран навіть вийшовши за рамки масиву, поки не зустрів би в якійсь комірці пам'яті символ \ 0. Можете спробувати підставити в перший приклад замість циклу оператор виведення рядку і побачите, що вийде. Наприклад **показало ось так:**

```
Text and stop!#Text and stop!
str=
```

Зверніть також увагу на відмінність символьної константи (в одинарних лапках - 'f', '@') від рядкової константи (в подвійних лапках "f", "@"). Для першої, компілятором С ++ виділяється один байт для зберігання в пам'яті. Для символу записаного в подвійних лапках, буде виділено два байта пам'яті - для самого символу і для нульового (додається компілятором).

Символьна константа складається з одного символу ASCII між апострофами ('). Приклади спеціальних символів:

Таблиця 4.1

Новий рядок	'\n'
Горизонтальна табуляція	'\t'
Повернення каретки	'\r'
Апостроф	'\''
Лапки	'\"'
Нульовий символ	'\0'
Зворотний слеш	'\'

Усі константи-рядки в тексті програми, навіть ідентично записані, розміщуються за різними адресами в статичній пам'яті. З кожним рядком пов'язується сталий покажчик на його перший символ. Власне, рядок-константа є виразом типу "покажчик на char" зі сталим значенням - адресою першого символу.

Так, присвоювання p="ABC" (*p* - покажчик на *char*) встановлює покажчик p на символ 'A'; значенням виразу *("ABC"+1) є символ 'B'.

Елементи рядків доступні через вказівники на них, тому будь-який вираз типу "вказівник на char" можна вважати рядком.

Необхідно мати також на увазі те, що рядок вигляду "x" - не те ж саме, що символ 'x'. Перша відмінність : 'x' - об'єкт одного з основних типів даних мови Сі (*char*), в той час, як "x" - об'єкт похідного типу (масиву елементів типу *char*). Друга різниця : "x" насправді складається з двох символів - символу 'x' і нуль-символу.

Функції введення рядків.

Прочитати рядок із стандартного потоку введення можна за допомогою функції *gets*(). Вона отримує рядок із стандартного потоку введення. Функція читає символи до тих пір, поки їй не зустрінеться символ нового рядка '\n', який генерується натисканням клавіші ENTER. Функція зчитує всі символи до символу нового рядка, додаючи до них нульовий символ '\0'.

Синтаксис :

```
char *gets(char *buffer);
```

Як відомо, для читання рядків із стандартного потоку введення можна використовувати також функцію *scanf*() з форматом *%s*. Основна відмінність між *scanf*() і *gets*() полягає у способі визначенні досягнення кінця рядка; функція *scanf*() призначена скоріше для читання слова, а не рядка. Функція *scanf*() має два варіанти використання. Для кожного з них рядок починається з першого не порожнього символу. Якщо використовувати *%s*, то рядок продовжується до (але не включаючи) до наступного порожнього символу (пробіл, табуляція або новий рядок). Якщо визначити розмір поля як *%10s*, то функція *scanf*() не прочитає більше 10 символів або ж прочитає послідовність символів до будь-якого першого порожнього символу.

Що якщо рядок повинен буде ввести користувач за допомогою клавіатури? В цьому випадку необхідно оголосити масив типу char із зазначенням його розміру достатнього для зберігання символів, що вводять, включаючи \ 0. Не забувайте про цей нульовий символ. Якщо вам треба зберігати 3 символи в масиві, його розмір повинен бути на одиницю більше - тобто 4.

```
#include <stdio.h>
#include <locale.h>
int main ()
{  printf("Input 0:");
       char str[20];
       int i=0;
       while((str[i]=getchar()) && str[i]!='\n' && i<19) {i++; }
       str[i] ='\0';
       puts(str);
     }
```

Використовуючи порожні лапки при ініціалізації, ми присвоюємо кожному елементу масиву значення \ 0. Таким чином рядок буде очищений від "сміття" інших програм. Навіть якщо користувач введе назву містить меншу кількість символів, наступний за назвою буде символ \ 0. Це дозволить уникнути небажаних помилок. У пам'яті цей рядок буде виглядати так:рядки в С ++, символьні масиви в С ++

Примітка. Нульовий символ - це не цифра 0; він не виводиться на друк і в таблиці символів ASCII має номер 0. Наявність нульового символу передбачає, що кількість комірок масиву повинна бути принаймні на одну більше, ніж число символів, які необхідно розміщувати в пам'яті. Наприклад, оголошення

char str[10];

передбачає, що рядок містить може містити максимум 9 символів.

Таким чином, **варіанти введення рядків на Сі:**

1) **scanf("%s",name);**

Зауважимо, що ця команда введе рядок лише до першого пробілу або роздільника. Для того щоб ввести далі за допомогою scanf можна використати іншу форму:

scanf("%[^\n]s",name); // цей варіант scanf ігнорує пробіли та нові рядки при вводі

Але в цьому варіанті можуть бути ще проблеми при існуванні попереднього вводу. Тоді можна або очистити буфер за допомогою fflush (stdin); або використати scanf з пробілом перед специфікатором

scanf(" %[^\n]s",name);

2) gets(siteName);

Ця форма має Buffer Overflow, тому краще скористатись наступною формою, де вказується довжина рядку:

fgets(siteName, 20, stdin);

При цьому потрібно вводити не більше символів ніж в розмірі буферу (в даному випадку - 20).

3) fscanf(stdin,"%s",name);

Файлова форма scanf() де ми на початку вказуємо стандартний файл введення з консолі.

4) Ще одна файлова форма введення для якої потрібно взначити змінну Name фіксованої довжини символів:

fread(Name, 1, sizeof(Name),stdin);

Виведення

Тепер розглянемо інші функції виведення рядків. Для виведення рядків можна використовувати функції *puts*() і *printf*().

Синтаксис функції puts():

int puts(char *string);

Ця функція виводить всі символи рядка *string* у стандартний потік виведення. Виведення завершується переходом на наступний рядок.

Синтаксис функції printf() :

printf(“%s”, string);

Різниця між функціями *puts*() і *printf*() полягає в тому, що функція *printf*() не виводить автоматично кожний рядок з нового рядка.

Стандартна бібліотека мови програмування Сі містить клас функцій для роботи з рядками, і всі вони починаються з літер str. Для того, щоб використовувати одну або декілька функції необхідно підключити файл string.h (#include<string.h>)

Можна використовувати динамічний масив символів для того, щоб працювати з рядками:

```
#include <stdio.h>
const int MAX = 4;
int main () {
char *names[] = {
"Zara Ali",
"Hina Ali",
"Nuha Ali",
"Sara Ali"
};

int i = 0;
for ( i = 0; i < MAX; i++) {
   printf("Value of names[%d] = %s\n", i, names[i] );
}
return 0;
}
```

результат роботи:

```
Value of names[0] = Zara Ali
Value of names[1] = Hina Ali
Value of names[2] = Nuha Ali
Value of names[3] = Sara Ali
```

Основні функції роботи з рядками

Визначення довжини рядка. Для визначення довжини рядка використовується функція *strlen*(). Її синтаксис :

size_t strlen(const char *s);

Функція *strlen*() повертає довжину рядка *s*, при цьому завершуючий нульовий символ не враховується.

Приклад :

```
char *s= "Some string";
int len;
```

Наступний оператор встановить змінну *len* рівною довжині рядка, що адресується вказівиком *s*:

```
len = strlen(s); /* len == 11 */
```

Копіювання рядків. Оператор присвоювання для рядків не визначений. Тому, якщо *s1* і *s2* - символьні масиви, то неможливо скопіювати один рядок в інший наступним чином.

```
char s1[100];
char s2[100];
s1 = s2; /* помилка Останній оператор (s1=s2;) не скомпілюється.*/
```

Щоб скопіювати один рядок в інший необхідно викликати функцію копіювання рядків *strcpy*(). Для двох покажчиків *s1* і *s2* типу char * оператор strcpy(s1,s2); копіює символи, що адресуються покажчиком *s2* в пам'ять, що адресується покажчиком *s1*, включаючи завершуючі нулі.

Для копіювання рядків можна використовувати і функцію *strncpy*(), яка дозволяє обмежувати кількість символів, що копіюються.

strncpy(destantion, source, 10);

Наведений оператор скопіює 10 символів із рядка *source* в рядок *destantion*. Якщо символів в рядку *source* менше, ніж вказане число символів, що копіюються, то байти, що не використовуються встановлюються рівними нулю.

Примітка. Функції роботи з рядками, в імені яких міститься додаткова літера n мають додатковий числовий параметр, що певним чином обмежує кількість символів, з якими працюватиме функція.

Конкатенація рядків.

Конкатенація двох рядків означає їх об'єднання, при цьому створюється новий, більш довгий рядок. Наприклад, при оголошенні рядка

```
char first[]= "Один ";
```

оператор

strcat(first, "два три чотири!");

перетворить рядок first в рядок "Один два три чотири".

При викликанні функції strcat(s1,s2) потрібно впевнитися, що перший аргумент типу char * ініціалізований і має достатньо місця щоб зберегти результат. Якщо s1 адресує рядок, який вже записаний, а s2 адресує нульовий рядок, то оператор strcat(s1,s2); перезапише рядок s1, викликавши при цьому серйозну помилку.

Функція *strcat*() повертає адресу рядка результату (що співпадає з її першим параметром), що дає можливість використати "каскад" декількох викликів функцій :

strcat(strcat(s1,s2),s3);

Цей оператор додає рядок, що адресує *s2*, і рядок, що адресує *s3*, до кінця рядка, що адресує *s1*, що еквівалентно двом операторам:

strcat(s1,s2);

strcat(s1,s3);

Повний список прототипів функцій роботи з рядками можна знайти в таблиці 4.2.

Порівняння рядків.

Функція *strcmp*() призначена для порівняння двох рядків. Синтаксис функції :

int strcmp(const char *s1, const char*s2);

Функція *strcmp*() порівнює рядки *s1* і *s2* і повертає значення 0, якщо рядки рівні, тобто містять одне й те ж число однакових символів. При порівнянні рядків ми розуміємо їх порівняння в лексикографічному порядку, приблизно так, як наприклад, в словнику. У функції насправді здійснюється посимвольне порівняння рядків.

Кожний символ рядка *s1* порівнюється з відповідним символом рядка *s2*. Якщо *s1* лексикографічно більше *s2*, то функція *strcmp*() повертає додатне значення, якщо менше, то - від'ємне.

Прототипи всіх функцій, що працюють з рядками символів, містяться у файлі string.h. У файлі <string.h> оголошена велика кількість функцій для роботи з рядками. Всі функції працюють з рядками, що закінчуються нульовим символом. Нижче наведено оголошення деяких функцій і їхнє призначення:

Таблиця 4.2

char *strcat (char * dest , const char *source). Функція дописує рядок dest у кінець рядка source;

char *strncat(char * dest, const char *source, unsigned n). Функція дописує не більш n символів рядка dest у кінець рядка source;

char *strchr (const char *source, int c) – пошук у рядку source першого входження зліва символу c, повертає покажчик на знайдений символ або NULL;

char strrchr (const char *source, int c) – пошук у рядку source першого входження зправа символу c, повертає покажчик на знайдений символ або NULL;

int strcmp (const char *s1, const char *s2) – порівнює рядки посимвольно, зліва направо. Повертає 0, якщо рядки s1 і s2 рівні, повертає негативне число, якщо s1 за алфавітом раніш s2, повертає позитивне число, якщо s1 за алфавітом пізніше s2.

int strncmp (const char *s1, const char *s2, unsigned n) – порівнює рядки за першими n символами;

int stricmp (const char *s1, const char *s2) – порівнює рядки без обліку регістра символів;

char *strcpy (char * dest, const char *source) – копіювання рядка source у рядок dest;

char *strncpy (char * dest, const char *source, unsigned n) – копіювання не більш n перших символів у рядок dest;

int strlen (const char *s) – повертає довжину рядка s;

char *strlwr (char *s) – перетворює символи рядка в нижній регістр (у маленькі букви);

char *strupr (char *s) – перетворює символи рядка у верхній регістр (у великі букви);

char *strset (char *s, int c) – заповнюють весь рядок s символом c;

char *strnset (char *s, int c, unsigned n) – заміняє перші n символів рядка s на символ c;

char *strrev (char *s) – розташовує всі символ рядка s, за винятком нуль термінатора, у зворотному порядку;

size_t strcspn (const char *s1, const char *s2) – повертає довжину початкової ділянки рядка s1, що складається із символів, яких немає в s2;

size_t strspn (const char *s1, const char *s2) – повертає довжину початкової ділянки рядка s1, що складається тільки із символів, що є в s2;

char *strpbrk (char *s1, char *s2) – переглядає рядок s1 зліва направо, поки не зустрінеться кожний із символів рядка s2; повертає покажчик на знайдений символ або NULL;

char *strtok (char *s1, const char *s2) – застосовується для послідовного пошуку в рядку s1 фрагментів рядка, обмежених символами з заданої множини *розділителів*. Множина розділителів задається рядком s2. Перше звертання до strtok фіксується у рядок (s1), в якому ведеться пошук. Для пошуку другого і третього і т.д. фрагментів рядка у тім же рядку в якості першого параметру задається NULL. Функція повертає покажчик на знайдений фрагмент рядка або NULL, якщо в рядку токенов не залишилося. Вихідний рядок змінюється – по мірі знаходження фрагментів рядка, перший за ними роздільник заміняється на нуль термінатор. Програма для експериментів з strtok наведена в додатку. Програма, в котрій за допомогою strtok виділяються слова в рядку, наведена у вказівках до лабораторної роботи.

char *strstr (const char *s1, const char *s2) – шукає перше з ліва входження рядка s2 у рядок s1. Повертає або покажчик на s2 у s1, або NULL, якщо s1 не містить рядка s2.

char *strdup (const char *s) – функція створює копію рядка s. Пам'ять для копії виділяється автоматичним викликом malloc(strlen(s) + 1). Програміст повинний самостійно звільнити цю пам'ять, коли вона стане непотрібною.

Робота з рядками, що не обов’язково закінчуються нулевим байтом

Для роботи з масивом символів, що не обов'язково має у кінці нульовий символ, можна користуватися функціями перетворення буферів. Прототипи цих функцій знаходяться у файлі string.h. Ці функції дозволяють присвоювати кожному байту в межах вказаного буфера задане значення, а також використовуються для порівняння вмісту двох буферів. Наприклад:

memcpy() — копіювання символів з одного буфера у другий, поки не буде скопійований заданий символ або не буде скопійовано визначену кількість символів

memcmp() — порівнює вказану кількість символів з двох буферів

У файлі ctype.h описано прототипи функцій, що призначені для перевірки літер. Ці функції повертають ненульове значення (істина), коли її аргумент задовольняє заданій умові або належить вказаному класу літер, та нуль в іншому випадку. Наприклад:

int islower(int c) — символ c є малою літерою;
int isupper(int c) — символ c є великою літерою;
int isalnum(int c) — символ c є буквою або цифрою;
int isalpha(int c) — символ c є буквою;
int tolower(int c) — перетворення літери у нижній регістр;
int strtol(int c) — перетворення рядку у довге ціле число;

Приклад

Очистити вираз від пробілів та зайвих символів зліва:

```
char *ltrim(char *str, const char *seps)
{
   size_t totrim;
   if (seps == NULL) {
      seps = "\t\n\v\f\r ";
   }
   totrim = strspn(str, seps);
   if (totrim > 0) {
      size_t len = strlen(str);
      if (totrim == len) {
         str[0] = '\0';
      }
      else {
         memmove(str, str + totrim, len + 1 - totrim);
      }
   }
   return str;
}
```

Таблиця 4.3

Методи класифікації символів

В заголовочному файлі <ctype.h>	
isalnum	символ є літерою або цифрою
isalpha	символ є літерою

islower	символ є малою літерою
isupper	символ є великою літерою
isdigit	символ є цифрою
isxdigit	символ є шістнадцятковою цифрою
iscntrl	символ є контрольним символом
isgraph	символ є графічним символом
isspace	символ є пробілом
isblank(C99)	символ є порожнім символом
isprint	символ є друкованим
ispunct	символ є пунктуаційним

Character manipulation

tolower	перетворення літери у нижній регістр
toupper	перетворення літери у верхній регістр

Примітка: додаткові функції, чиї імена починаються з або до, або в нижній регістрі, можуть бути додані до заголовка ctype.h і не повинні визначатися програмами, які включають цей заголовок.

В бібліотеці stdlib.h визначаються функції для перетворення рядків у числові типи.

Таблиця 4.4

Перетворення у рядків у числові типи

Заголовочний файл <stdlib.h>	
atof	переведення рядку до дійсного типу
atoi, atol, atoll(C99)	переведення рядку до цілого значення
strtol, strtoll(C99)	переведення рядку до цілого значення
strtoul, strtoull(C99)	переведення рядку до натурального значення
strtof, strtod (C99), strtold(C99)	переведення рядку до дійсного типу
Заголовочний файл <inttypes.h>	
strtoimax(C99), strtoumax(C99)	переведення рядку до intmax_t та uintmax_t

Функції роботи з рядками

Заголовочний файл <string.h>	
strcpy, strcpy_s(C11)	копіює рядки один в інший
strncpy, strncpy_s(C11)	копіює визначену кількість символів з одного рядку в інший
strcat, strcat_s(C11)	конкатенує два рядки
strncat,strncat_s(C11)	копіює визначену кількість символів двох рядків
strxfrm	ттрансоформує рядок, щоб strcmp мав той же

результат, що й strcoll

Інормація по рядкам

Заголовочний файл <string.h>

strlen(), strnlen_s(C11)	довжина рядку
strcmp	порівняння двох рядків
strncmp	порівнює визначену кількість символів в рядку
strcoll	порівняння двох рядків в даній локалі
strchr	знаходить першу позицію символу в рядку
strrchr	знаходить останню позицію символу в рядку
strspn	повертає довжину максимального сегменту в одному рядку що збігається з символами в другому рядку
strcspn	повертає довжину максимального сегменту в одному рядку що не містяться у символах в другого рядку
strpbrk	знаходить першу позицію хоч якогось символу з першого рядку в другому рядку
strstr	знаходить першу позицію підрядку
strtok, strtok_s(C11)	знаходить наступний роздільник в рядку

Операції з памяттю

Заголовок <string.h>

memchr	знаходить першу позицію символу в байтовому масиві
memcmp	порівнює два буфери
memset, memset_s(C11)	заповнює буфер символом
memcpy, memcpy_s (C11)	копіює два буфери
memmove, memmove_s (C11)	пересуває два буфери

Різне

Визначено в <string.h>

strerror, strerror_s(C11) strerrorlen_s (C11)	повертає текстову версію даної помилки

https://purecodecpp.com/uk/archives/920