**Покажчики та масиви**

Припустимо нескінченно велике число різних типів покажчиків і масивів. Далі йдуть типові приклади.

***Покажчик на основний тип***:

char \*p;

*Змінна р є покажчиком на символ, тобто цій змінній повинна присвоюватися адреса символу*.

***Покажчик на покажчик***:

char \*\*t;

*Змінна t - покажчик на покажчик символу*.

***Одновимірний масив***:

int a[50];

*Змінна* а - *масив з 50 цілих чисел*.

***Двомірний масив***:

char m[7][50];

*Змінна* m - *масив з семи масивів, кожен з яких складається з 50 символів*.

*Масив з семи покажчиків*:

char \*r[7];

*Масив* r складається з покажчиків на символи.

***Покажчик на функцію***:

int (\*f)();

f - *покажчик на функцію, що повертає ціле значення*.

**Масиви**

*Масив* є складним об'єктом, що складається з об'єктів-компонентів, які називаються елементами одного і того ж типу. Прості визначення *масиву* мають вид

Тип данных x[n1][n2]...[nk]

Де x - ідентифікатор, який визначається як ім'я масиву, а ni - ***розмірності масиву***. *Масив* x зветься ***k-мірним массивом*** з елементами типу тип даних. Елементи i -го виміру мають індекси від 0 до ni-1. Тип елемента масиву може бути одним з основних типів, типом іншого масиву, типом покажчика (pointer), типом структури ( struct ) або типом об'єднання ( union ). Хоча елементи масиву не можуть бути функціями, вони можуть бути покажчиками на функції. Нижче наведені деякі приклади визначень масиву:

int page[10]; /\* одновимірний масив з 10

элементів, перенумерували з 0 до 9 \*/

char line[81];

float big[10][10], sales[10][5][8]; /\*двумірний

масив и трьохмірний масив\*/

Посилання на елемент k-мірного масиву x робляться за допомогою наступного позначення:

x[i1][i2]...[ik]

де ij - цілий вираз, при цьому 0<=ij<=nj-1, а nj - максимальне значення j -го індексу масиву x. Наприклад:

page[5]

line[i+j-1]

big[i][j]

Вказуючи тільки перші p індексів, можна посилатися на k-p - мірний подмассів k-мірного масиву (p<=k), Наприклад,

sales[i] /\* посилання на двовимірний підмасив масиву

sales \*/

sales[i][j] /\* посилання на одновимірний підмасив \*/

sales[i][j][k] /\* посилання на елемент масиву \*/

**Покажчики**

**Покажчиком** називається компонент заданого типу, що є посиланням на деяку область пам'яті. Визначення покажчика має такий вигляд:

тип-даних \*id1, \*id2, \*\_, \*idn

Тип змінних id1, id2, \_, *idn* визначається як тип покажчиків на тип-даних. Ці змінні служать посиланнями на об'єкти типу тип-даних. Цей тип називається базовим типом змінних-покажчиків. Нижче наведено кілька прикладів визначень покажчиків:

int \*pi, \*qi;/\* покажчики на цілі об'єкти \*/

char \*c; /\* покажчик на символьний об'єкт \*/

**Динамічні об'єкти**

Указатели используются при создании и обработке *динамических объектов*. Заранее определяемые объекты создаются с помощью определений. ***Динамічні об'єкти***, на відміну від заздалегідь визначених, створюються динамічно і явно в процесі виконання програми. Для створення динамічних об'єктів служать функції malloc та calloc. На відміну від заздалегідь визначених об'єктів, число динамічних об'єктів не фіксоване тим, що записано в тексті програми, - за бажанням динамічні об'єкти можуть створюватися і знищуватися в процесі її виконання. Динамічні об'єкти, на відміну від заздалегідь визначених, не мають імен, і посилання на них виконується за допомогою покажчиків.

*Значення* 0 може бути присвоєно вказівниками будь-якого типу. Це значення показує, що даний покажчик не містить посилання на який-небудь об'єкт. Спроба використовувати це значення для звернення до об'єкта може привести до помилки, але тільки в операційних системах із захистом пам'яті. За угодою, для позначення константи з нульовим значенням використовується ідентифікатор NULL, опис якого знаходиться в бібліотеці stddef.h і є сістемозавісімим.

Створення динамічних об'єктів

За стандартом аргументи функцій malloc, calloc мають тип об'єкта, що повертається void \*.

char \*s = (char\*)malloc(size);

unsigned size; /\* обсяг пам'яті, який необхідно виділити \*/

char \*s = (char \*)calloc(nelem,elsize);

unsigned nelem; /\* число елементів, для яких потрібно виділити пам'ять \*/

unsigned elsize; /\* обсяг пам'яті, який необхідно виділити для кожного елемента \*/

/\* або просто замінивши char\* на void\* \*/

void\* calloc(nelem, elsize);

unsigned nelem;

unsigned elsize;

Обидві функції повертають покажчик на виділену пам'ять. Для визначення необхідного обсягу пам'яті можна використовувати оператор sizeof:

sizeof (выражение)

Обсяг пам'яті, необхідний для зберігання вираження:

sizeof(T)

Обсяг пам'яті, необхідний для зберігання значень типу T.

Функції malloc і calloc повертають покажчик на створений динамічний об'єкт. Фактично функції повертають знакові покажчики, які можуть бути явно перетворені до потрібного типу покажчика. Значення, повернуті функціями розподілу пам'яті, використовуються для посилань на динамічні об'єкти. Наприклад, за допомогою оператора

pi = (int \*) malloc(sizeof(int));

виділяється пам'ять для одного цілого значення. Адреса цієї області пам'яті присвоюється змінній pi після його перетворення з типу char \* (покажчик на знак), з яким він повертається функцією malloc, до типу int \* (покажчик на ціле), тобто типу змінної pi.

*Доступ до динамічних об'єктів*

Присвоєння значення об'єкту, посилання на який задана покажчиком pi, виконується за допомогою імені покажчика \* pi, наприклад:

\*pi = 55;

Одне і те ж значення може бути присвоєно більш ніж одній змінній-вказівнику. Таким чином, можна посилатися на динамічний об'єкт за допомогою більш одного покажчика. Про об'єкт, до якого можна звертатися з використанням більш ніж одного покажчика, кажуть, що він має псевдоімена (alias). Наприклад, в результаті присвоювання qi = pi;

### і qi, і pi вказують на один і той же об'єкт, тобто вони є псевдоіменамі. Некероване використання псевдоімен може завдати шкоди розуміння тексту програми, так як можливість доступу до одного і того ж об'єкту і його модифікація за допомогою різних псевдоімен не завжди очевидні при аналізі частини програми.

**Час життя динамічного об'єкта**

Пам'ять, яка займаеться динамічними об'єктами, якщо вона необхідна для інших цілей, повинна бути звільнена явним зазначенням. В іншому випадку ця пам'ять може бути втрачена, тобто стане неможливим її повторне використання. Явна звільнення виконується використанням функції free, яка має наступну специфікацію:free(ptr)

char \*ptr;

Необхідно вживати заходів обережності для уникнення помилок, пов'язаних з посиланнями на об'єкт, пам'ять для якого вже звільнена - проблема висячого посилання (Horowwitz, E. 1983. Fundamentals of Programming Languages. Computer Science Press).

Якщо реалізація мови забезпечує збірку сміття, то пам'ять, яку займає об'єктами, до яких можна отримати доступ, може бути автоматично підготовлена для повторного використання. Однак в мові Сі, на відміну від мов Лисп і Снобол, така можливість відсутня.

**Вказівка на заздалегідь визначені об'єкти**. Покажчики можуть забезпечувати посилання на заздалегідь визначені об'єкти. Адреса такого об'єкта може бути визначений використанням оператора адресації & (address of operator). Наприклад, розглянемо змінні i і pi, визначені як

int i, \*pi;

Присвоєння

pi = &i;

pi дозволяє посилатися на об'єкт з ім'ям i також за допомогою покажчика pi, використовуючи позначення \* pi. Імена i і \* pi - псевдоімена. Оператор & є також стандартним засобом моделювання передачі параметрів по посиланню. Однак його вживання може призвести до проблеми висячого посилання.

**Вказівка на довільну комірку пам'яті**. За допомогою явних перетворень можна отримати покажчик на довільну комірку пам'яті. Наприклад, припустимо, що pt є покажчиком типу T \*. Тоді покажчик на комірку пам'яті 0777000 можна отримати за допомогою наступного запису:

pt = (T\*)0777000;

Звернення до конкретних елементів пам'яті часто буває необхідно в програмах, які взаємодіють з обладнанням, наприклад в драйверах пристроїв, коли для управління пристроями потрібно мати доступ до таких осередків пам'яті, як регістри стану або осередку буфера пристрою. Хоча такі можливості корисні і навіть необхідні для деяких додатків, користуватися ними слід з обережністю.

**Зв'язок між покажчиками і масивами**

У мові Сі масиви і покажчики тісно пов'язані. Ім'я кожного масиву може розглядатися як покажчик на перший елемент масиву. Елемент масиву a [i] є елемент масиву, на який вказує значення a + i, тобто \* (A + i), де значення а є адресою першого елемента масиву а, а саме a [0]. Вираз a + i є прикладом арифметичних дій з покажчиками - ціле значення i складається зі значенням покажчика, адресою першого елемента масиву а. Значення цього виразу є а плюс обсяг пам'яті, займаний i елементами масиву a. Припустимо, що x - двовимірний масив. Тоді посилання на подмассів x [i] є посиланням на i-й рядок масиву x. x [i] дає адресу першого елемента цього рядка, тобто \* (X + i). Елементи кожного рядка займають безперервну область пам'яті, так як масиви зберігаються записаними по рядках, тобто при запису елементів масиву в пам'ять швидше за всіх змінюється останній індекс.

Аналогічно, посилання на y [i], де y - n-мірний (n> 1) масив, є посиланням на (n-1) -мірний підмасив з елементами y [i, j2, j3, \_jn], де значення jk відповідають визначенню масиву y. y [i] дає адресу першого елемента цього підмасива, тобто \* (Y + i). Всі елементи цього (n-1) -мірного підмасива займають безперервну область пам'яті.

**Рядки - додаткові відомості про тісний зв'язок між покажчиками і масивами**

***Рядки*** - це масиви знаків. За угодою, останнім знаком рядка повинен бути нульовий знак \ 0. Оскільки ім'я масиву фактично є покажчиком на перший елемент масиву, змінні типу string можуть також розглядатися, як такі, що тип char \*. Наприклад, друга змінна string\_array у визначенні.

char \*string\_pointer, string\_array[81];

може розглядатися також як знаковий покажчик. Для рядка, представленої першої змінної string\_pointer, пам'ять повинна бути виділена явно. З іншого боку, для масиву string\_array пам'ять є покажчиком на неї. Зауважимо, що пам'ять повинна бути також виділена або зарезервована для ознаки кінця рядка \ 0.

|  |  |
| --- | --- |
| ! | Немає нічого незвичайного не тільки в *інтерпретації змінних* типу string, тобто *масивів* знаків як покажчиків, а й в інтерпретації рядків, які також можуть розглядатися двояко - як *масиви* і як покажчики - і все в одній програмі! Це особливо важливо, коли *рядки* передаються як аргументи функції. Викликаюча програма може розглядати *рядок* як *масив* символів, а викликається, може розглядати її як знаковий покажчик. Якщо довжина *рядка* непостійна, то використання знакових покажчиків для *рядків* має певні переваги. Хоча *рядки* змінної довжини можуть бути також реалізовані з використанням *масивів*, така реалізація виявляється занадто неекономною з точки зору використання пам'яті і накладає обмеження на максимальну довжину *рядка*. Наприклад, для розміщення *рядків* різної довжини може бути створений *масив* знакових покажчиків. Альтернативне рішення з використанням двовимірного масиву знаків в загальному випадку буде використовувати пам'ять неефективно, так як в цьому випадку треба було б зробити число стовпців дорівнює кількості знаків в рядку найбільшою можливою довжини. |

**Ініціалізація масивів і класи пам'яті**

Ми знаємо, що скалярні змінні можна ініціалізувати в описі типу за допомогою таких виразів, як наприклад:

int fix = 1;

float flax = PI\*2;

при цьому передбачається, що PI - раніше введене макроозначення. Чи можна ініціалізувати *масиви*?

Зовнішні, статичні і автоматичні масиви можна ініціалізувати!

Реєстрові масиви ініціалізувати не можна!

Якщо нічого не засилати в *масив* перед початком роботи з ним, то зовнішні, статичні та автоматичні *масиви* не ініціалізуються для числових типів нулем і '\ 0' (null) для символьних типів, а реєстрові *масиви* містять якесь сміття, що залишилося в цій частині пам'яті. Якщо в статичному, зовнішньому або автоматичному *масиві* нам потрібні початкові значення, відмінні від нуля, в цьому випадку ми можемо робити так:

/\* дні місяця \*/

int days[12]={31,28,31,30,31,30,31,31,30,31,30,31};

main( )

{

int index;

extern int days[];/\*необов'язковий опис \*/

for(index = 0; index<12; index++)

printf("Місяць %d має %d днів.\n", index+1,

days[index]);

}

Результат:

Місяць 1 має 31 днів.

Місяць 2 має 28 днів.

Місяць 3 має 31 днів.

Місяць 4 має 30 днів.

Місяць 5 має 31 днів.

Місяць 6 має 30 днів.

Місяць 7 має 31 днів.

Місяць 8 має 31 днів.

Місяць 9 має 30 днів.

Місяць 10 має 31 днів.

Місяць 11 має 30 днів.

Місяць 12 має 31 днів.

Кількість елементів у списку ініціалізації має відповідати розміру *масиву*. Якщо список менше розміру *масиву*, то елементи *масиву*, на яких не вистачило списку, будуть забиті нулями. Якщо ж список більше *масиву*, то компілятор видасть синтаксичну помилку. Треба просто виділити *масив*, розмір якого буде достатній для розміщення списку.

Предыдущую программу лучше переписать так:

int days[ ] = {31,28,31,30,31,30,31,31,30,31,30,31};

main( )

{

int index;

extern int days[ ];/\* необов'язковий опис \*/

for(index=0;index<sizeof(days)/(sizeof(int));

index++)

printf("Місяць %d має %d днів.\n",index +1,

days[index]);

}

До цієї програми слід зробити два суттєвих зауваження.

**Перше**: якщо ми використовуємо порожні дужки для ініціалізації *масиву*, то компілятор сам визначить кількість елементів у списку і виділить для нього *масив* потрібного розміру.

**Друге**: воно стосується додавання, зробленого в керуючому операторі for. Не покладаючись на свої обчислювальні можливості, ми поклали завдання підрахунку розміру *масиву* на *компілятор*. Оператор sizeof визначає розмір в байтах об'єкта або типу, наступного за ним. Припустимо в нашій системі розмір кожного елемента типу int дорівнює двом байтам, тому для отримання кількості елементів *масиву* ми ділимо загальне число байтів, займане *масивом*, на 2. Проте в інших системах елемент типу int може мати інший розмір. Тому в загальному випадку виконується розподіл на значення змінної sizeof для елемента типу int.

В результаті роботи цієї програми ми отримуємо точно 12 значень. Наш метод, що дозволяє програмі самій знаходити розмір *масиву*, не дозволив нам надрукувати кінець *масиву*.

### Функції, масиви і покажчики

*Масив*и можна використовувати в програмі двояко. По-перше, їх можна описати в тілі функції. По-друге, вони можуть бути аргументами функції. Все, що було сказано про *масиви*, відноситься до першого їх застосування. Тепер розглянемо масиви в якості аргументів. Проаналізуємо скелет програми, звертаючи увагу на описи.

/\* масив-аргумент \*/

main( )

{

int ages[50]; /\* масив з 50 элементів \*/

convert(ages);

\_

}

convert(years);

int years[ ];/\* який розмір масиву? \*/

{

\_

}

Очевидно, що *масив* ages складається з 50 елементів. А що можна сказати про *масив* years? Виявляється в програмі немає такого масиву. Описувач

int years[ ];

створює не *масив*, а *покажчик* на нього! Подивимося, чому це так. Ось виклик нашій функції:

convert(ages);

ages - *аргумент функції* convert. Ім'я ages є *покажчиком* на перший елемент *масиву*, що складається з 50 елементів. Таким чином, оператор виклику функції передає їй покажчик, т. е. адреса функції convert (). Це означає, що аргумент функції є покажчиком, і ми можемо написати функцію convert () наступним чином:

convert(years);

int \*years;

{

\_

}

Дійсно, *оператори*

int years[ ];

int \*years;

- синоніми. Обидва вони оголошують змінну years покажчиком *масиву* цілих чисел. Однак головна їхня відмінність полягає в тому, що перший з них нагадує нам, що *покажчик* years посилається на *масив*.

Як тепер зв'язати його з масивом ages? При використанні покажчика в якості аргументу, функція взаємодіє з відповідною змінною викликаючій програмі, тобто оператори, що використовують *покажчик* years в функції convert () , фактично працюють з *масивом* ages, що знаходяться в тілі функції main () . Коротше кажучи, коли ім'я *масиву* застосовується в якості аргументу, функції передається покажчик. Потім функція використовує цей покажчик для виконання змін у вихідному *масиві*, який належить програмі, що викликає функцію.

### Операції з покажчиками

1. **Привласнення**. Вказівнику можна привласнити адресу. Зазвичай ми виконуємо цю дію, використовуючи ім'я *масиву* або операцію отримання адреси &.
2. **Визначення значення**. Операція \* видає значення, що зберігається в зазначеній комірці.
3. **Отримання адреси покажчика**. Подібно будь-яким змінним, змінна типу покажчик має адресу і значення. Операція & повідомляє нам, де знаходиться сам покажчик.
4. **Збільшення покажчика**. Ми можемо виконувати цю дію за допомогою звичайної операції додавання або за допомогою операції збільшення. Збільшуючи *покажчик*, ми переміщаємо його на наступний елемент *масиву*.
5. **Різниця**. Можна знаходити різницю двох покажчиків. Зазвичай це робиться для покажчиків, що посилаються на елементи одного і того ж *масиву*, щоб визначити, на якій відстані один від одного знаходяться елементи. Пам'ятайте, що результат має той же тип, що і змінна, що містить розмір *масиву*!