Міністерство освіти і науки України

Національний університет “Львівська Політехніка”

Лабораторна робота №17

Виконав:
Студент групи АП-11
Білий Анатолій Іванович

Прийняв:

Чайковський І.Б.

Львів – 2024

“Дослідження особливостей використання вказівників у мові С”

Мета роботи: ознайомитися з поняттям вказівник та особливостями його використання у процесі програмування.

Теоретичні відомості

Вказівник – це змінна, значенням якої є адреса деякого об'єкта (зазвичай іншої змінної) в пам'яті комп'ютера. Наприклад, якщо одна змінна містить адресу іншої змінної, то говорять, що перша змінна вказує (посилається) на другу.

Важлива особливість мови С полягає в тому, що вказівник в ній типізовані. Це означає,що якщо змінна, скажімо, **p** має тип «вказівник на int», то значеннями змінної **p** можуть бути адреси лише змінних типу int та не можуть бути адреси змінних типу double чи інших.

Взагалі, який би тип в мові С не розглянути, йому відповідає свій тип вказівника. Щоб оголосити змінну типу вказівника на деякий тип, потрібно перед іменем змінної поставити зірочку:

*ім ' я_типу * ім ' я _ змінної _ вказівника ;*

В одному оголошенні можна поєднувати звичайні змінні та змінні типу вказівника. Наприклад:

```
int  *p ;
double    *q1,*q2   ;
int x , *r, s = 0 , *t ;
```

В першому рядку оголошено змінну p типу вказівника на ціле. В другому рядку оголошено одразу дві змінні типу вказівника на дійсні числа: значеннями змінної q1 та змінної q2 можуть бути адреси змінних типу double. В третьому рядку оголошено змінну x типу цілого числа, потім змінну r типу вказівника на ціле, далі змінну s знову звичайного цілого типу (з одночасним присвоєнням початкового значення) і, нарешті, змінну t типу вказівника на ціле.

Операція взяття адреси & знаходить адресу змінної. Нехай **a** – ім'я деякої змінної, тоді вираз &a дає адресу змінної **a**. Наприклад, після оператору присвоєння

*int *p , x = 10;*

p = &x ;

значенням змінної **p** стане адреса змінної **x**: вираз в правій частині визначає адресу змінної **x**, а оператор присвоювання присвоює отриману адресу у змінну **p**. Тепер, як кажуть, змінна-вказівник **p** показує на змінну **x**.

Присвоювання вказівників. Вказівник можна використовувати в правій частині оператора присвоювання для присвоювання його значення іншому вказівнику. Якщо обидва вказівники мають один і той же тип, то виконується просте присвоювання, без перетворення типу. Розглянемо таку ситуацію на прикладі.

```
#include <stdio.h>
int main(void)
{
 int x = 99;
 int *p1, *p2;
 p1 = &x;
 p2 = p1;
 /* друк значення x два рази */
 printf("Znachennya po adresi p1 i p2: %d %d\n", *p1, *p2);
 printf("Znachennya vkazivnukiv p1 i p2: %p %p", p1, p2);
 return 0;
}
```

Після присвоювання

p1 = & x;

p2 = p1;

обидва вказівники (p1 і p2) посилаються на x. Тобто, обидва вказівники посилаються на один і той же об'єкт, і відповідно мають однакове значення у наведеній програмі.

Зверніть увагу, що для виведення значень вказівників в функції printf () використовується специфікатор формату % p, який виводить адреси в форматі, використовуваному компілятором.

Операції адресної арифметики для вказівників. У мові С допустимі тільки дві арифметичні операції над вказівниками: сумування і віднімання.

Припустимо, що поточне значення вказівника p1 типу int * є 2000. Припустимо також, що змінна типу int займає в пам'яті 2 байти. Тоді після

операції збільшення p1 ++; вказівник p1 приймає значення 2002 а не 2001. Тобто, при збільшенні на 1 вказівник p1 буде посилатися на наступне ціле число. Це ж справедливо і для операції зменшення.

Наприклад, якщо p1 дорівнює 2000, то після виконання оператора p1--; значення p1 дорівнюватиме 1998.

Операції адресної арифметики підкоряються наступним правилам. Після виконання операції збільшення над вказівником, даний вказівник буде посилатися на наступний об'єкт свого базового типу. Після виконання операції зменшення - на попередній об'єкт. Стосовно вказівників на char, операції адресної арифметики виконуються як звичайні арифметичні операції, тому що довжина об'єкта char завжди дорівнює 1. Для всіх вказівників адреса збільшується або зменшується на величину, яка дорівнює розміру об'єкта того типу, на який вони вказують. Тому вказівник завжди посилається на об'єкт з типом, тотожним базовому типу вказівника.

Операції адресної арифметики не обмежені збільшенням і зменшенням. Наприклад, до вказівників можна додавати цілі числа або віднімати з них цілі числа.

Виконання оператора

p1 = p1 + 12;

"пересуває" вказівник p1 на 12 об'єктів в бік збільшення адрес.

Крім підсумовування і віднімання покажчика і цілого, дозволена ще тільки одна операція адресної арифметики: можна віднімати два вказівника. Завдяки цьому можна визначити кількість об'єктів, розташованих між адресами, на які вказують два вказівника; правда, при цьому вважається, що тип об'єктів збігається з базовим типом вказівників.

Всі інші арифметичні операції заборонені, наприклад не можна ділити і множити вказівники, виконувати над вказівниками побітові операції тощо.

Порівняння вказівників. Стандартом С дозволяється порівняння двох вказівникв. Наприклад, якщо оголошені два вказівники p і q, то наступний оператор є правильним:

if (p <q) printf ("p посилається на меншу адресу, ніж q \ n");

Як правило, порівняння вказівників може виявитися корисним, тільки тоді, коли два вказівники посилаються на загальний об'єкт, наприклад, на масив. Як приклад розглянемо програму з двома стековими функціями, призначеними для запису і зчитування цілих чисел.

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 5
void push(int i);
int pop(void);
int  *tos,*p1,stack[SIZE];
int main(void)
{
 int value;
 tos = stack; /* tos посилається на основу стеку */
 p1 = stack; /* ініціалізація p1 */
 do {
   printf("Vvedit znachennya: ");
   scanf("%d", &value);
   if(value != 0) push(value);
   else printf("Znachennya na vershuni rivne %d\n", pop());
 } while(value != -1);
 return 0;
}
void push(int i)
{
 p1++;
 if(p1 == (tos+SIZE)) {
   printf("Perepovnennya steka.\n");
   exit(1);
 }
 *p1 = i;
}
int pop(void)
{
 if(p1 == tos) {
   printf("Stek pyctuy.\n");
   exit(1);
 }
 p1--;
 return *(p1+1);
}
```

Стек - це список, який використовує систему доступу "першим увійшов - останнім вийшов". Іноді стек порівнюють зі стопкою тарілок на столі: перша, поставлена на стіл, буде взята останньою. Стеки часто використовуються в

компіляторах, інтерпретаторах, програмах обробки великоформатних таблиць і в інших системних програмах.

Для створення стека необхідні дві функції: push () і pop (). Функція push () заносить числа в стек, а pop () - витягує їх. В даному прикладі ці функції використовуються в main (). При введенні числа з клавіатури, програма поміщає його в стек. Якщо ввести 0, то число вилучають із стека. Програма завершує роботу при введенні -1.

Стек зберігається в масиві stack. Спочатку вказівники p1 і tos встановлюються на перший елемент масиву stack. Надалі p1 посилається на верхній елемент стека, а tos продовжує зберігати адреси основи стеку. Після ініціалізації стека використовуються функції push () і pop (). Вони виконують запис в стек і зчитування з нього, перевіряючи кожен раз дотримання границі стека. У функції push () перевіряється, що вказівник p1 не перевищує верхньої границі стека tos + SIZE. Це запобігає переповненню стека.

У функції pop () перевіряється, що вказівник p1 не виходить за нижню границю стека. В операторі return функції pop () дужки необхідні тому, що без них оператор return * p1 + 1; повернув би значення, розташоване за адресою p1, збільшене на 1, а не значення за адресою p1 + 1.

Вказівники і масиви. Поняття вказівників і масивів тісно пов'язані. Розглянемо наступний фрагмент програми:

*char str [80], * p1;*

p1 = str;

Тут p1 вказує на перший елемент масиву str. Звернутися до п'ятого елементу масиву str можна за допомогою будь-якого з двох виразів:

str [4]

** (p1 + 4)*

Масив починається з нуля. Тому для п'ятого елемента масиву str потрібно використовувати індекс 4. Можна також збільшити p1 на 4, тоді він буде вказувати на п'ятий елемент. У мові С існують два методи звернення до елементу масиву: адресна арифметика та індексація масиву.

Стандартний запис масивів з індексами наочний і зручний у використанні, проте за допомогою адресної арифметики іноді вдається скоротити час доступу до елементів масиву. Тому адресна арифметика часто використовується в програмах, де істотну роль грає швидкодія.

У наступному фрагменті програми наведено дві версії функції putstr (), що виводить рядок на екран. У першій версії використовується індексація масиву, а в другій - адресна арифметика:

```
 / * Індексація вказівника s як масиву. * /
void putstr (char * s)
{
  register int t;
  for (t = 0; s [t]; ++ t) putchar (s [t]);
}
/ * Використання адресної арифметики. * /
void putstr (char * s)
{
   while (*s) putchar (*s ++);
}
```

Більшість програмістів вважають другу версію більш наочною і зручною. Для більшості компіляторів вона також більш швидкодіюча. Тому в процедурах такого типу прийоми адресної арифметики використовуються досить часто.

Масиви вказівників. Як і об'єкти будь-яких інших типів, вказівники можуть бути зібрані в масив. У наступному операторі оголошений масив з 10 вказівників на об'єкти типу int:

```
int * x [10];
```

Для присвоєння, наприклад, адреси змінної var третьому елементу масиву вказівників, необхідно написати:

```
x [2] = & var;
```

В результаті цієї операції, такий вираз приймає те ж значення, що і var:

```
* x [2]
```

Масиви вказівників часто використовуються при роботі з рядками. Наприклад, можна написати функцію, що виводить потрібний рядок з повідомленням про помилку за індексом num:

```
void syntax_error (int num)
{
  static char * err [] = {
    "Не можна відкрити файл \ n",
    "Помилка при читанні \ n",
    "Помилка при записі \ n",
    "Неякісний носій \ n"
  };
  printf ( "% s", err [num]);
}
```

Масив err містить вказівники на рядки з повідомленнями про помилки. Тут рядкові константи у виразі ініціалізації створюють вказівники на рядки. Аргументом функції printf () служить один з вказівників масиву err, який відповідно до індексу num вказує на потрібний рядок з повідомленням про помилку. Наприклад, якщо в функцію syntax_error () передається num зі значенням 2, то виводиться повідомлення Помилка при записі.

Хід роботи

1. Ознайомитися з теоретичними відомостями.

2. Здійснити виконання прикладів, представлених у теоретичних відомостях, після чого представити скріни їх коду та результати виконання у звіті.

3. Написати программу для визначення суми масиву, який складається з п’яти елементів. Значення елементів вводяться з клавіатури. Для доступу до елементів масиву використати вказівник. Представити скріни коду програми та результати її виконання у звіті.

4. Оформити звіт.

Приклад 1

```
#include <stdio.h>
int main(void){
int x = 99;
```

```
int *p1, *p2;
p1 =&x;
p2= p1;
printf("Значення по адресі p1 ip2: %d %d\n", *p1, *p2);
printf("значення вказівника p1 і p2: %p %p", p1, p2);
return 0;
}
```

```
Значення по адресі p1 ip2: 99 99
значення вказівника p1 і p2: 0x7fff878f9164 0x7fff878f9164
```

Приклад 2

```
// Приклад 2
#include <stdio.h>
#include <stdlib.h>
#define SIZE 5
void push(int i);
int pop(void);
int *tos,*p1,stack[SIZE];
int main(void)
{
   int value;
   tos = stack; // tos посилається на основу стеку
   p1 = stack; // ініціалізація p1
   do {
      printf("Ведіть значення:");
      scanf("%d", &value);
      if(value != 0)
         push(value);
      else
         printf("Значення на вершині рівне %d\n", pop());
   } while(value != -1);
   return 0;
}

void push(int i)
{
   p1++;
   if(p1 == (tos+SIZE)) {
      printf("Perepovnennya steka.\n");
      exit(1);
   }
   *p1 = i;
}
int pop(void)
{
   if(p1 == tos) {
```

```
        printf("Stek pyctuy.\n");
        exit(1);
    }
    p1--;
    return *(p1+1);
}
```

```
Ведіть значення:99
Ведіть значення:-1
```

Приклад 3

```
#include <stdio.h>
void main(){
    int a[5];
    int sum=0;
    for (int i=0;i<5;i++){
        scanf("%d",&a[i]);
    }
    for(int j=0;j<5;j++){
        int c = *(a+j);
        sum = sum + c;
    }
    printf("sum = %d",sum);
}
```

```
5
6
3
5
5
sum = 24
```

Приклад 4

```
#include <stdio.h>
// Функція для обчислення суми додатних елементів масиву
int sum_positive_elements(int arr[], int size) {
int sum = 0;
for (int i = 0; i < size; i++) {
if (arr[i] > 0) {
sum += arr[i];}
}
return sum;
}
int main() {
int arr[10]; // Масив розмірністю 10
int size = 10; // Розмір масиву
// Введення елементів масиву з клавіатури
printf("Введіть 10 цілих чисел:\n");
```

```
for (int i = 0; i < size; i++) {
printf("Елемент %d: ", i + 1);
scanf("%d", &arr[i]);}
// Виклик функції для обчислення суми додатних елементів
int sum = sum_positive_elements(arr, size);
// Виведення результату
printf("Сума додатних елементів масиву: %d\n", sum);
return 0;}
```

```
Введіть 10 цілих чисел:
Елемент 1: 1
Елемент 2: 2
Елемент 3: 3
Елемент 4: 4
Елемент 5: 5
Елемент 6: 6
Елемент 7: 7
Елемент 8: 8
Елемент 9: 9
Елемент 10: 10
Сума додатних елементів масиву: 55
```

Завдання

```
#include <stdio.h>
void main(){
    int a[5];
    int sum=0;
    for (int i=0;i<5;i++){
        scanf("%d",&a[i]);
    }
    for(int j=0;j<5;j++){
        int c = *(a+j);
        sum = sum + c;
    }
    printf("sum = %d",sum);
}
```

```
2
1
4
2
4
sum = 13
```

Контрольні питання

1. Дайте визначення поняття вказівник.

Вказівник у мові програмування C — це змінна, яка зберігає адресу

іншої змінної. Вказівники є потужним інструментом, що дозволяє працювати з динамічною пам'яттю, створювати складні структури даних, передавати великі об'єкти функціям ефективно та маніпулювати масивами і рядками.

2. Які арифметичні операції можуть виконуватись з вказівниками?

Арифметичні операції, які можуть виконуватися з вказівниками в мові програмування C, включають:

1. **Додавання числа до вказівника**:
   - Можна додати ціле число до вказівника, що зміщує його на вказану кількість елементів. Наприклад, якщо **p** є вказівником на масив, то **p + 1** вказуватиме на наступний елемент масиву.
2. **Віднімання числа від вказівника**:
   - Можна відняти ціле число від вказівника, що зміщує його назад на вказану кількість елементів. Наприклад, **p - 1** вказуватиме на попередній елемент масиву.
3. **Віднімання одного вказівника від іншого**:
   - Можна відняти один вказівник від іншого, якщо вони вказують на елементи одного і того ж масиву. Це дає кількість елементів між ними. Наприклад, якщо **p** вказує на **arr[5]**, а **q** вказує на **arr[2]**, то **p - q** дорівнює **3**.
4. **Інкрементування (++)**:
   - Операція інкрементування збільшує вказівник на один елемент. Наприклад, **p++** зміщує вказівник **p** на наступний елемент масиву.
5. **Декрементування (--)**:
   - Операція декрементування зменшує вказівник на один елемент. Наприклад, **p--** зміщує вказівник **p** на попередній елемент масиву.

3. Поясніть призначення функції pop () у стеку.

Функція pop() у структурі даних стек призначена для видалення та повернення верхнього елемента зі стека. Стек працює за принципом LIFO (Last In, First Out), тобто останній доданий елемент буде першим видаленим. Функція pop() зменшує розмір стека на один елемент.

4. Які є методи звертання до елементу масиву?

У мові програмування С є два основні методи звертання до елементів масиву:

Індексація:

Це найпоширеніший спосіб доступу до елементів масиву. Індекс масиву починається з нуля, тобто перший елемент масиву має індекс 0, другий - 1 і так далі.

Синтаксис: array[index].

Наприклад, якщо є масив int arr[5] = {10, 20, 30, 40, 50};, то arr[2] звертається до третього елемента масиву (значення 30).

Вказівники:

У С масиви тісно пов'язані з вказівниками. Ім'я масиву є вказівником на перший елемент масиву.

Можна використовувати арифметику вказівників для доступу до елементів масиву.

Синтаксис: *(array + index).

Наприклад, якщо є масив int arr[5] = {10, 20, 30, 40, 50};, то *(arr + 2) також звертається до третього елемента масиву (значення 30).

5. Які переваги використання прийомів адресної арифметики при зверненні до елементів масиву?

Використання адресної арифметики при зверненні до елементів масиву надає кілька переваг. Воно дозволяє гнучко маніпулювати покажчиками для роботи з різними частинами масиву або динамічною пам'яттю, що може бути ефективніше за рахунок зменшення кількості обчислень. Також це забезпечує уніфікований спосіб роботи з різними типами даних і структурами, полегшуючи реалізацію складних структур даних, таких як зв'язані списки, дерева та графи. Крім того, адресна арифметика спрощує обхід масиву за допомогою інкрементування або декрементування покажчика.

Висновок: під час виконання цієї лабораторної роботи я ознайомився з

поняттям вказівник та особливостями його використання у процесі програмування.