Лекція 19

Масиви як параметри функцій. Класи пам’яті.

Способи передачі даних в функції

У мові С ++ дані в підпрограму можна передавати трьома способами: за значенням, за адресою та за посиланням. У мові С допустимі тільки два способи: за значенням і за адресою.

Передача даних за значенням

Цей спосіб передачі даних в підпрограму є основним і діє по-замовчуванню. Фактичний параметр обчислюється в викликаючій функції і його значення передається на місце формального параметра в функції, що викликається. На цьому зв'язок між фактичним і формальним параметрами припиняється.

В якості фактичний параметр можна використовувати константу, змінну або більш складний вираз. Передача даних за значенням придатна тільки для простих даних, які є вхідними параметрами. Якщо параметр є вихідним даним або масивом, то передача його в функцію за значенням не прийнятна.

Приклад 1. Обчислити суму ряду із заданою точністю $\varepsilon = 10^{-5}$:

$$s = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \dots$$

Для обчислення суми ряду використовуємо функцію. До неї передамо за значенням ***x*** і ***eps***. Результат повернемо через ім'я функції оператором **return**.

Можливий варіант реалізації програми:

```
#include <iostream>
using namespace std;
#include <cmath>

double fsum(double x, double eps);

int main()
{ double x, s, eps = 1.0e-5;
  cout << "x = "; cin >> x;
  s = fsum(x, eps);
  cout << "s = " << s << endl;
  return 0;}

double fsum(double x, double eps)
{ double s = x, p = x, i, t = x * x;
  for(i = 3; fabs(p) > eps; i += 2)
  { p = -p * t / (i * (i - 1));
    s += p; }
  return s;}
```

Через ім'я функції можна повернути тільки один об'єкт, інші доведеться повертати через список. Цей спосіб (передача за значенням) не дозволяє повернути через список параметрів змінені значення.

Передача даних за адресою

За адресою в функцію завжди передаються масиви (розглянемо це в наступних темах). Для масиву це взагалі єдиний спосіб передачі даних в мовах C/C ++. Так само за адресою можна передати ті прості об'єкти, які є вихідними даними (або вхідними та вихідними одночасно).

У разі передачі даних за адресою фактичний параметр може бути тільки змінної (константа або вираз не мають адреси!).

Повернемося до попереднього прикладу.

Приклад. Дано два числа, що зберігаються в змінних ***a*** і ***b***. Використовуючи підпрограму, виконати обмін вмісту осередків цих змінних.

Дані передамо за адресою. Вони будуть в цьому завданні і вхідними, і вихідними даними. Для контролю зміни вмісту комірок пам'яті будемо виводити на екран монітора проміжні дані.

Можливий варіант реалізації програми:

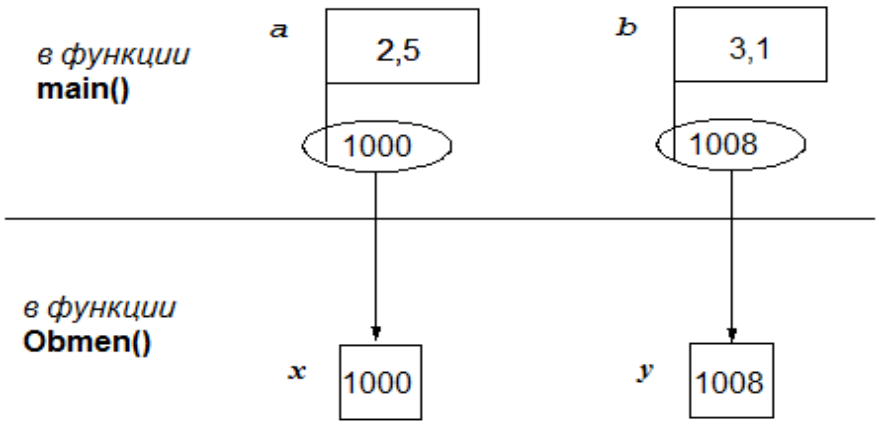
```
#include <iostream>
using namespace std;
void Obmen(double *x, double *y);
int main()
{ double a = 2.5, b = 3.1;
  cout << "Do Obmen: a=" << a << " b=" << b << endl;
  Obmen(&a, &b);
  cout << "Posle Obmen: a=" << a << " b=" << b << endl;
  return 0;}

void Obmen(double *x, double *y)
{ double c;
  cout << "Function Obmen start:\n *x=" << *x << " *y=" << *y << endl;
  c = *x;
  *x = *y;
  *y = c;
  cout << "Function Obmen end:\n *x=" << *x << " *y=" << *y << endl;}
```

Результат

```
Do Obmen: a=2.5 b=3.1
Function Obmen start:
*x=2.5 *y=3.1
Function Obmen end:
*x=3.1 *y=2.5
Posle Obmen: a=3.1 b=2.5
```

Друк на екрані монітора показує, що обмін проведений, і вихідні змінні тепер мають нові значення, тобто передача даних за адресою дійсно дозволяє повернути у функцію результат роботи підпрограми, що викликається.

Як це працює? Розглянемо це питання докладніше, використовуючи приклад з обміном даних. Для наочності наведемо малюнок:



У викликаючій функції (в нашому випадку – в **main()**) обчислюються адреси об'єктів, переданих за адресою (у нас – адреси змінних ***a*** і ***b***. Нехай це будуть числа 1000 і 1008), і потім ці адреси копіюються в осередку пам'яті – покажчики, пам'ять під які виділено в функції **Obmen()** (це ***x*** і ***y***). Знаючи адресу змінної, наприклад, адреса змінної ***a***, тепер зберігається в покажчику ***x***, можна, користуючись операцією розіменування, не тільки прочитати, але й змінити значення вихідної змінної.

Ні якої реальної передачі даних (в сенсі копіювання) з підпрограми **Obmen()** назад в **main()** не робиться. Ми насправді через покажчики працюємо з вихідними об'єктами! Тому після виходу з функції **Obmen()** маємо змінені змінні ***a*** і ***b*** (якщо бути точніше, змінні зміняться ще до виходу з функції, тобто в момент перестановки в самій функції **Obmen()**).

Передача даних по посиланню

Це ще один із способів повернути результат роботи функції через список параметрів. Нагадаємо, що застосовується тільки для С ++. У мові С такого варіанту немає.

При передачі даних по посиланню в функцію, куди передаються дані, створюються синоніми вихідних об'єктів. Тому робота в підпрограмі ведеться саме з вихідними об'єктами. Якщо в підпрограмі посилальна змінна змінить значення, то це відразу позначиться на вихідної змінної.

У викликаючій функції параметр, який передається по посиланню, може бути тільки простою змінної будь-якого відомого типу.

Повернемося знову до прикладу обміну, тільки дані передамо по посиланню.

Приклад. Дано два числа, що зберігаються в змінних ***a*** і ***b***. Використовуючи підпрограму, виконати обмін вмісту комірок цих змінних.

Можливий варіант реалізації програми:

```
#include <iostream>
using namespace std;
void Obmen(double &x, double &y);
int main()
{ double a = 2.5, b = 3.1;
  cout << "Do Obmen: a=" << a << " b=" << b << endl;
  Obmen(a, b);
  cout << "Posle Obmen: a=" << a << " b=" << b << endl;
```

```
   return 0;}

void Obmen(double &x, double &y)
{ double c;
   cout << "Function Obmen start:\n x=" << x << " y=" << y << endl;
   c = x;
   x = y;
   y = c;
   cout << "Function Obmen end:\n x=" << x << " y=" << y << endl;}
```

Результат

```
Do Obmen: a=2.5 b=3.1
Function Obmen start:
x=2.5 y=3.1
Function Obmen end:
x=3.1 y=2.5
Posle Obmen: a=3.1 b=2.5
```

Зверніть увагу на те, що для об'єкта, переданого за посиланням, в списку формальних параметрів вказується значок **&** перед ім'ям змінної (працюємо з посиланням на об'єкт), а далі в тілі підпрограми використовується просто ім'я цієї змінної. Коли виконується виклик, в списку фактичних параметрів задається ім'я потрібної змінної.

Для простих об'єктів передача даних по посиланню краще, ніж передача за адресою, так як в цьому випадку текст функції простіше, легше читається, не потрібно виконувати операції розіменування.

До недоліку способу (передача даних по посиланню) можна віднести те, що за викликом функції не можна зрозуміти, що параметр передається саме за посиланням, а не за значенням, і як наслідок, цей параметр швидше за все змінюється функцією. Щоб не було сумнівів, завжди дивіться на запис прототипів функцій, використаних в програмі. Прототипи завжди доступні, навіть якщо самі функції є тільки у вигляді об'єктних модулів.

Масиви як параметри функцій

Передача в функцію одновимірного масиву

Використання масиву як параметра функції не викликає ніяких труднощів. Як було сказано раніше, масив в підпрограму завжди передається за адресою, тому досить при виконанні функції вказати адресу початку масиву або адреса того елемента, починаючи з якого передбачається обробляти масив.

Приклад. Написати програму, яка буде викликати функцію для виведення на друк елементів масиву.

Можливий варіант рішення:

```
#include <iostream>
using namespace std;
void print(int x[], int n);
int main()
{ const int n = 5;
   int x[n] = {3, 5, 1 ,7, 4};
   print(x, n);
```

```
  return 0;}
void print(int x[], int n)
{ cout << "Massiv:" << endl;
  for(int i = 0; i < n; i++)
    cout << x[i] << endl;}
```

При виконанні функції оператором **print(x, n);** в неї передається адреса початку масиву ***x*** і кількість елементів ***n***. Виклик еквівалентний запису **print(&x[0], n);,** так як ім'я масиву це адреса на його початок. Таке трактування дозволяє передавати масив не починаючи з деякого номера.

Матриця як параметр функції

Матриця також передається за адресою. При виклику функції достатньо вказати ім'я матриці (адреса початку масиву). У списку функції, що викликається, масив описується так:

Тип_даних Им'я[][Константа]

Приклад. Написати і протестувати функцію виведення значень цілочисельної матриці на екран монітора.

Можливий варіант реалізації

```
#include <iostream>
using namespace std;
#define M 3
void PrintMatr(int a[][M], int n, int m);
int main()
{ const int n=2, int m=M;
  int a[n][m] = {{5, 4, -1}, {2, 6, 7}};
  PrintMatr(a, n, m);
  return 0;}
// Вывод матрицы на экран монитора
void PrintMatr(int a[][M], int n, int m)
{ int i, j;
  cout << "Matriza:" << endl;
  for(i = 0; i < n; i++)
  {  for(j = 0; j < m; j++)
       cout << a[i][j] << " ";
    cout << endl; }
}
```

Зручна така передача матриці в підпрограму? У загальному випадку ні. Це прийнятно тільки для невеликих програм з матрицями однакового розміру. Для кількох матриць різного розміру можна дати граничну кількість рядків (***M***) по максимуму.

Хорошим рішенням може стати використання динамічного масиву. В цьому випадку в підпрограму буде передаватися покажчик на покажчик. Ніяких глобальних констант не буде потрібно.

Наведемо приклад програми з використанням динамічної матриці і підпрограм, в які ця матриця передається.

Приклад. Дана прямокутна матриця. Перетворити матрицю за правилом: переставляємо місцями перший стовпець і останній, потім – другий і передостанній, і т.д.

```
#include <iostream>
using namespace std;
void ReadMatr(int **a, int n, int m);
void PrintMatr(int **a, int n, int m);
void P(int **a, int n, int m);
int main()
{ int n=3, m=4, i;
  int **a;
  a = new int*[n];
  for(i = 0; i < n; i++)  a[i] = new int [m];
  ReadMatr(a, n, m);
  P(a, n, m);
  PrintMatr(a, n, m);
  for(i = 0; i < n; i++)  delete []a[i];
  delete []a;
  return 0;}

void ReadMatr(int **a, int n, int m) // Ввод матрицы с клавиатуры
{ int i, j;
  cout << "Input matriza A(" << n << "*" << m << "):" << endl;
  for(i = 0; i < n; i++)
    for(j = 0; j < m; j++)    cin >>a[i][j];}

void PrintMatr(int **a, int n, int m) // Вывод матрицы на экран монитора
{ int i, j;
  cout << "Matriza:" << endl;
  for(i = 0; i < n; i++)
    { for(j = 0; j < m; j++)   cout << a[i][j] << " ";
    cout << endl; } }

void P(int **a, int n, int m) // Перестановка столбцов
{ int i, j, j1, c;
  for(i = 0; i < n; i++)
    for(j = 0, j1 = m - 1; j < j1; j++, j1--)
    { c = a[i][j];
      a[i][j] = a[i][j1];
      a[i][j1] = c;  }  }
```

Недоліком використання в якості параметрів динамічних масивів (будь-яких: одновимірних або багатовимірних) є зниження швидкості виконання програми, тому що додатково потрібен час на виділення і звільнення динамічної пам'яті під масиви. До того ж, необхідно не забувати звільняти динамічну пам'ять.

Багатовимірні масиви як параметри функцій

Для багатовимірних масивів (маються на увазі масиви тривимірні, чотиривимірні і так далі) ми маємо ті ж проблеми, що і для матриць. Якщо передавати, наприклад, звичайний тривимірний масив в підпрограму, то прототип функції може виглядати так:

void Print(int V[][N][M], int k, int n, int m);

Тут необхідні вже дві константи: ***N*** і ***M***.

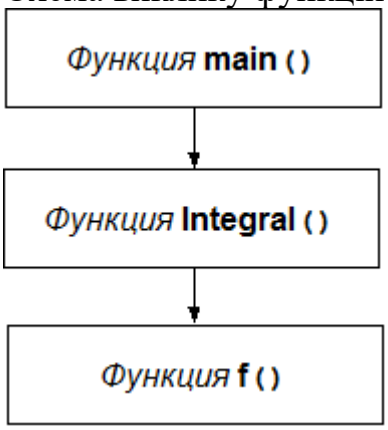
Рішення проблеми – так само в використанні динамічних масивів:
void Print(int ***V, int k, int n, int m);

Передача імені функції в підпрограму

Через список параметрів в підпрограму можна передавати не тільки дані і(або) адреси областей пам'яті, де зберігаються будь-які дані, але і адреси інших функцій.

Розглянемо це н прикладі обчислення визначеного інтегралу. Нехай нам необхідно створити універсальну функцію обчислення визначеного інтегралу для будь-якої підінтегральної функції.

Схема виклику функцій наведена на рис.



Виходячи з того, що кожне ім’я до своєї появи повинно бути описане, то послідовність опису прототипів повинна бути наступною:

1) опис прототипу функції;
2) опис прототипу інтегрування.

Програма може мати наступну реалізацію (формула трапецій):

```
#include <iostream>
using namespace std;
double f(double x);
double Integral(double a, double b, int n);

int main()
{ int n = 20;
  double a = 1, b = 2;
  double y = Integral(a, b, n);
  cout << "y="<< y << endl;
  return 0;}

// Функция для вычисления определённого интеграла по методу трапеций
double Integral(double a, double b, int n)
{ double s = (f(a) + f(b))/2, h = (b - a) / n, x = a + h;
  for(int i = 1; i < n; i++, x += h)
    s += f(x);
  s *= h;
  return s;}

double f(double x) // Подынтегральная функция
{
```

```
  return x*x;}
```

Біль цікава задача, коли необхідно знайти, наприклад суму двох інтегралів:

```
#include <iostream>
#include <cmath>
using namespace std;
double f1(double x);
double f2(double x);
double Integral(double (*f)(double x), double a, double b, int n);

int main()
{ int n = 20;
  const double PI=3.14159265;
  double a = -1, b = PI/2;
  double y = Integral(f1, a, 0, n) + Integral(f2, 0, b, n);
  cout << "y=" << y << endl;
  return 0;}

// Функция для вычисления определённого интеграла по методу трапеций
double Integral(double (*f)(double x), double a, double b, int n)
{ double s = (f(a) + f(b))/2, h = (b - a) / n, x = a + h;
  for(int i = 1; i < n; i++, x += h)
    s += f(x);
  s *= h;
  return s;}

// Подынтегральные функции
double f1(double x)
{
  return -x*x;}
double f2(double x)
{
  return sin(x);}
```

У прототипі функції **Integral()** і, природно, в її заголовку при визначенні першим параметром записано:

double (* f) (double x)

Перше слово **double** – це тип результату підінтегральної функції.

(* F) – формальне позначення імені переданої в підпрограму функції (це **f**). Перед ним записаний символ ***** (зірочка), що означає, що передається адреса функції. Дужки обов'язкові, інакше символ ***** (зірочка) буде відноситься до першого слова, тобто **double**.

(**double x**) – список формальних параметрів функції, яка передається в функцію **Integral()**. У нас є один формальний параметр типу **double**, який передається за значенням.

Класи пам’яті

Всі змінні в програмі характеризуються не тільки типом, але і класом пам'яті. У мові С існує чотири класи пам'яті: автоматичний (**automatic** – за замовчуванням), регвстровий (**register**), статичний (**static**) і зовнішній (**external**).

Автоматичні змінні (auto)

Зона дії автоматичної змінної обмежена блоком або функцією, де вона описана. Вона починає існувати після звернення до функції і зникає після виходу з неї. Таким чином автоматичні змінні не займають область в пам'яті. Значення автоматичної змінної не може бути змінено іншими функціями і в цих функціях може знаходиться змінні з таким же ім'ям. Змінну як автоматичну можна не описувати (приймається за замовчуванням).

Зовнішні змінні (external)

Зовнішні змінні вводяться як щось протилежне автоматичним. Це глобальні змінні і до них можна звертатися іменами з будь-якої функції. Оскільки pjdysiys змінні доступні скрізь, їх можна використовувати для зв'язку між функціями, не нехтуючи механізму формальних параметрів.

Приклад.

```
#include <stdio.h>
int x=145;/*Описание внешней переменной*/
main()
{extern int x,y;
   printf("x=%d y=%d \n",x,y);}
int y=541; /*Описание внешней переменной*/
```

Зовнішні змінні можуть визначаться поза функцією; при цьому виділяється фактична пам'ять. У будь-який інший функції, що звертається до цих змінним, вони повинні описуватися; робиться це з допомогою слова **extern**.

Статичні змінні (static)

Статичні змінні, подібно автоматичним, локальні в тій функції або блоці, де вони описані. Різниця полягає в тому, що статичні змінні не зникають, коли функція (блок) завершує роботу, і їх значення зберігаються для подальших викликів функції.

Розглянемо приклад, в якому змінна оголошена як статична.

```
*статические переменные*/
#include <stdio.h>
plus1()
{ static int x=0;
x=x+1;
printf("x=%d\n",x);}
  main()
{plus1();
plus1();
plus1();}
```

Результат: 1 2 3.

Регістровие змінні (register)

Регістровие змінні оголошуються в програмі за допомогою ключового слова **register** і за задумом автора мови С повинні зберігається в понад

швидкій пам'яті ЕОМ – регістрах. Використовуються аналогічно автоматичним змінним.