**Лекція 18**

## Формальні і фактичні параметри.

***Підпрограма*** – це частина всієї програми, оформлена особливим чином. Як правило, у вигляді підпрограми записується якась логічно завершена частина програми. Активне використання підпрограм при розробці програмного забезпечення складає основу модульного програмування.

***Модульне програмування*** – така технологія програмування, коли алгоритм всієї розв'язуваної задачі розбивається на окремі, логічно завершені частини. Якщо ці частини складні для кодування, то вони знову розбиваються на більш прості частини. Цей процес триває до тих пір, поки не вийдуть прості для програмування алгоритми, призначені для реалізації нескладних дій. Основні принципи модульного програмування були сформульовані ще в 60-х роках 20-го століття.

Головна відмінність підпрограми від основної (головної) програми полягає в тому, що управління може бути передано тільки головній програмі. Підпрограма може бути відкомпільована, але не може бути запущена на виконання.

Переваги від використання підпрограм:

- Можливість створення досить великих програм (обмеження – близько 50000 рядків. Розробка програм практично необмеженого розміру вимагає застосування класів. Мова про них піде далі).
- Досить просто повторно використовувати раніше написаний код.
- Розробку окремих частин програми, тобто підпрограм, можна доручити різним людям.
- Скорочується термін розробки програми в цілому за рахунок повторного використання коду і завдяки можливості залучення до програмного проекту цілої групи програмістів.
- Підвищується надійність програми, тому що підпрограми, як правило, не великі, їх можна досконально вивчити, до того ж повторне використання в нових програмах підпрограм, які вже застосовувалися раніше і не давали «збоїв», підвищує надійність нової програми в цілому.
- У ряді випадків зменшується розмір програми: якщо, наприклад, для сортування п'яти масивів, що використані в програмі, застосована одна і та ж підпрограма, а не пишеться практично один і той же код для кожного масиву окремо, то економія буде суттєвою. Звичайно, якщо підпрограма використовується в програмі тільки один раз, то розмір програми в цілому не тільки не скорочується, а навпаки, зростає.

Недоліки від застосування підпрограм:

- Використання підпрограм завжди зменшує швидкість роботи програми. Це стає помітним, коли розмір підпрограми занадто малий, наприклад, один-два оператора.
- Розмір вихідного коду і виконуваного модуля не завжди, але часто зростає при використанні підпрограм.

- Грамотне застосування підпрограм вимагає більш високої кваліфікації від програміста, ніж робота без підпрограм.

**Види підпрограм**

Існує дві категорії підпрограм: процедури і функції.

Процедура – це підпрограма, яка не повертає через своє ім'я результату роботи, тому вона викликається як окремий оператор. Процедура «спілкується із зовнішнім світом» через список параметрів. Частина з них будуть вхідними, частина – вихідними, тобто результатом роботи.

Функція – це підпрограма, яка через ім'я повертає результат своєї роботи. Функція викликається в вираженні, а не як окремий оператор. Через список параметрів вона може отримати вхідні дані і повернути результати роботи.

Традиційно підпрограму оформляють у вигляді функції, якщо результатом роботи є одиночний об'єкт (число, символ, рядок). Приклад алгоритмів, які зручно оформити функцією: обчислення значення певного інтеграла, знаходження мінімального числа в масиві, підрахунок кількості прогалин у рядку і т.д.

Якщо результатом роботи підпрограми є кілька об'єктів, або вихідними даними є масив, то необхідно використовувати процедуру, так як через ім'я підпрограми можна повернути тільки один простий об'єкт (число, адреса, символ), а через список параметрів – будь-яка кількість і одиночних і складених об'єктів . Повернення результату одночасно і через ім'я, і через список параметрів поганий, так як вводить користувача підпрограми в деяку оману щодо того, що може змінити підпрограма.

**Функції на мові C/C ++**

Формально в мовах C/C ++ немає процедур, а є тільки функції. Якщо ж необхідна саме процедура, то досить створити функцію, яка повертає тип void.

Розглянемо яким чином застосувати свою функцію в програмі на мові C ++. Щоб використовувати в програмі свою власну функцію, необхідно виконати три дії:

- задати прототип функції;
- викликати функцію в необхідному місці, наприклад, в функції main ()
- дати визначення функції.

Розберемося трохи докладніше з тим, що ми перерахували.

***Прототип функції*** – це заголовок функції, який закінчується крапкою з комою. Прототипи всіх функцій, які використовуються в програмі, зазвичай записують на початку тексту програми до визначення функцій. Прототип необхідний для того, щоб у компілятора була повна інформація про тип результату роботи функції і про список параметрів, переданих у функцію. Коли в тексті програми зустрінеться звернення до функції, то компілятор перевіряє правильність її виклику, звіряючись з інформацією з прототипу.

Для програміста прототипи теж корисні: можна в стислій формі відразу побачити, які функції використовуються в програмі, які параметри їм необхідні і т.д.

Виклик функції можливий у будь-який функції, де це буде потрібно. Якщо тип результату функції – void, то виклик записується окремим оператором, в інших випадках – у виразі того ж типу, що і тип результату функції. Мови C/C ++ дозволяють записати виклик функції у вигляді окремого оператора навіть в тому випадку, коли вона повертає тип результату, відмінний від void. Наприклад, оператор **sin(x);** абсолютно законний, хоча навряд чи варто так робити.

Визначення функції складається з її заголовка і тіла, записаного у вигляді блоку. Припустимо записувати визначення функцій в будь-якій послідовності. Не можна визначати функцію всередині іншої функції (C/C ++ – це не Паскаль!).

**Приклад.** Знайти максимум з двох чисел. Алгоритм пошуку максимуму оформити у вигляді функції.

Можливий варіант програми:

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

double fmax(double a, double b); // 1)Прототип функции

int main()
{ double a = 4, b = 3, max;
  cout << "a=" << a << " max=" << b << endl;
  max = fmax(a, b);    // 2)Вызов функции

  cout << "max=" << max << endl;
  return 0;}

double fmax(double x, double y) // 3)Определение функции
{ double max;
  if(x > y)
    max = x;
  else
    max = y;
  return max;}
```

**Формальні і фактичні параметри**

У роботі з підпрограмами доводиться мати справу з двома видами параметрів: список фактичних параметрів і список формальних параметрів.

***Список параметрів*** (фактичних або формальних – все одно) записується в круглих дужках відразу за ім'ям функції. В принципі, список параметрів може бути порожнім, але дужки за ім'ям функції обов'язкові!

***Список фактичних параметрів*** – це ті реальні дані, за допомогою яких можна налаштувати алгоритм підпрограми на обробку конкретних даних. Фактичні параметри вказуються в списку параметрів при виклику функції і передаються в цю функцію. У нашому прикладі при виконанні функції **fmax()** в списку фактичних параметрів вказані дві змінні: ***a*** і ***b***. Це реальні об'єкти, під які в функції **main()** виділена пам'ять, вони можуть мати якісь значення і т.д.

***Список формальних параметрів*** – це по суті набір вимог, які пред'являє підпрограма до переданих в неї даними. Список фактичних параметрів записується в заголовку функції при її визначенні в дужках. Для кожного параметра необхідно вказати тип і ім'я. При необхідності задається і спосіб передачі (про це пізніше – в наступній темі). Імена формальних параметрів локалізовані в підпрограмі і доповнюють перелік локальних об'єктів цієї підпрограми. Так, в нашій функції **fmax()** формальні параметри ***x*** і ***y*** типу **double** доповнюють список локальних змінних, що складається з однієї змінної ***max*** типу **double**. У результаті функція **fmax()** має три локальних змінних, відомих тільки в цій функції.

Між списками фактичних і формальних параметрів необхідно витримувати повну відповідність за кількістю параметрів, їх типу, порядку проходження і способу передачі в функцію. Недотримання цих вимог в кращому випадку (для програміста) призводить до помилки на стадії компіляції, в гіршому – до «Багів» програми, які будуть час від часу проявлятися під час виконання програми. «Виловити» ж такого роду помилки дуже непросто. Як правило, вони залишаються в програмі протягом всього періоду її використання. Тому дуже важливо дотримуватися правильність передачі даних в підпрограму. в функції **main()** виділена пам'ять, вони можуть мати якісь значення і т.д.

**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;
```

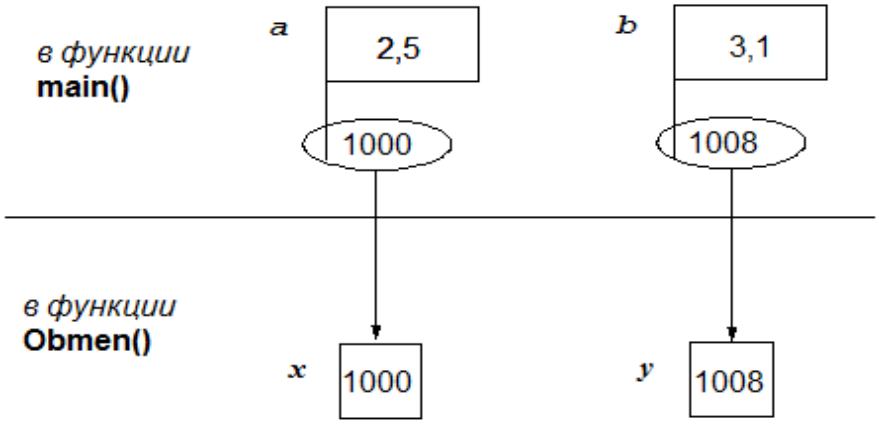
```
  cout << "Function Obmen end:\n *x=" << *x << " *y=" << *y << endl;}
```

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 імя 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:

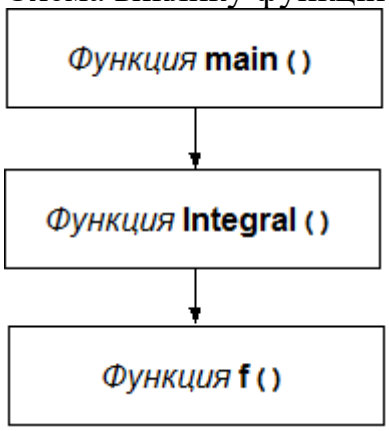
**void Print(int ***V, int k, int n, int m);**

**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.