Лекція 19

Рекурсія.
Активізація та запис оперативної пам'яті

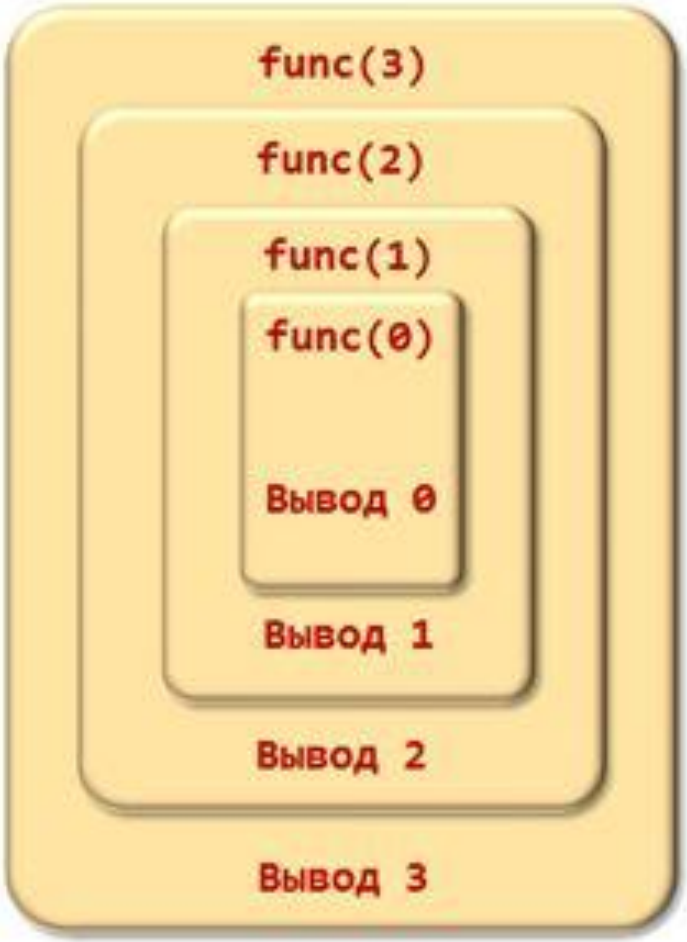
Поняття рекурсії

- ***Рекурсія*** - полягає у визначенні, описі, зображенні будь-якого об'єкта або процесу всередині самого цього об'єкта або процесу. Це ситуація, коли об'єкт є частиною самого себе.

- Процедура або функція може містити виклик інших процедур або функцій. У тому числі процедура може викликати саму себе. Комп'ютер лише послідовно виконує команди і, якщо зустрічається виклик процедури, просто починає виконувати цю процедуру. Без різниці, яка процедура дала команду це робити.

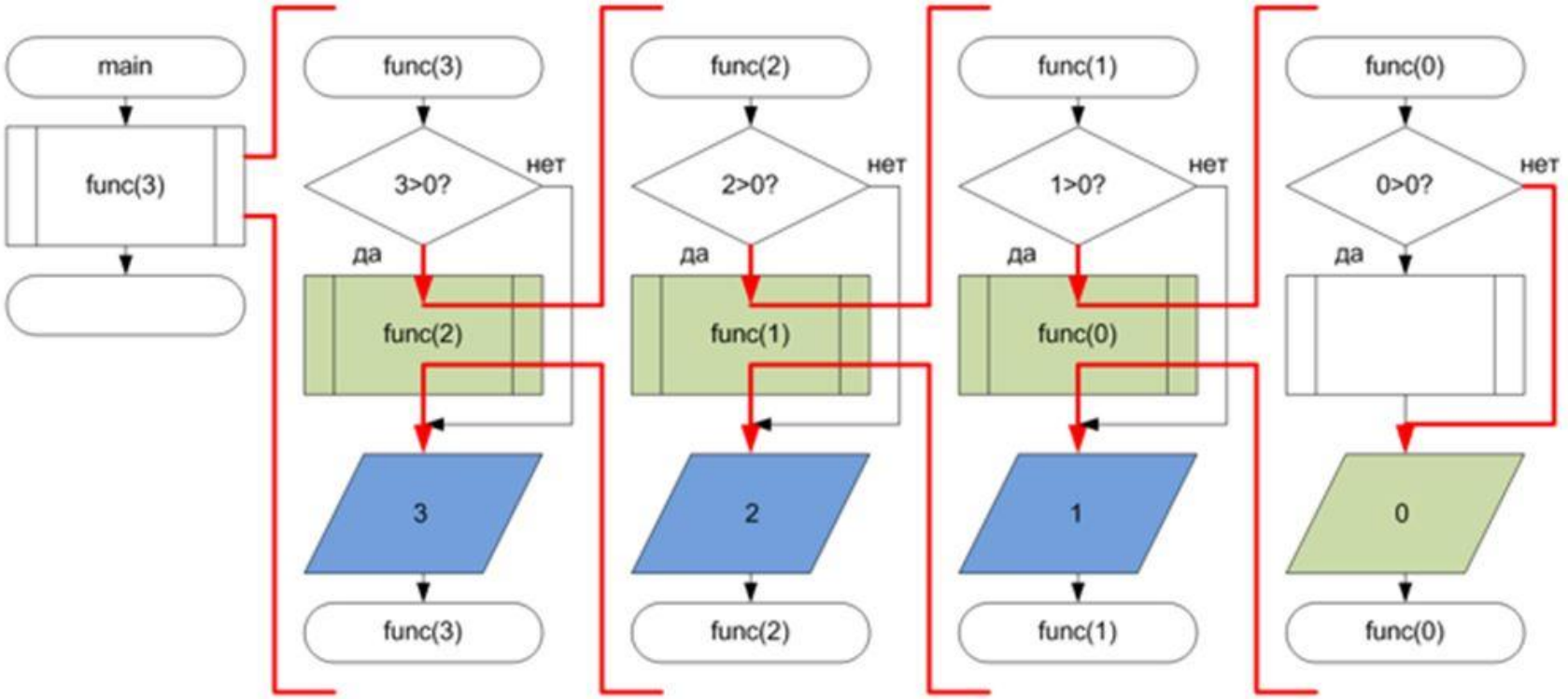
Поняття рекурсії

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

void func(int num)
{
  if (num > 0) func(num - 1);
  cout << num << " ";
}

int main()
{
  func(3);
  cin.get();
 return 0;
}
```



Поняття рекурсії

- Кількість одночасно виконуваних процедур називають ***глибиною рекурсії.***

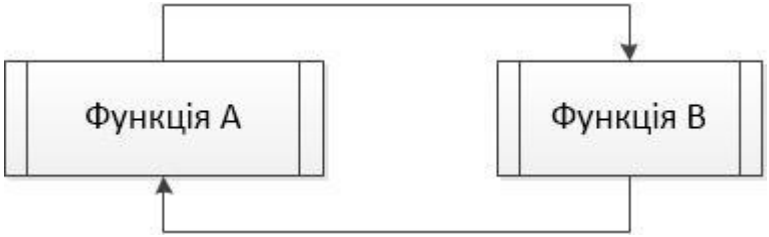


Поняття рекурсії

- Важливим і обов'язковим моментом у формуванні рекурсивної процедури є ***базис рекурсії***.
- **Базіс рекурсії визначає умова виходу з рекурсії.** Як правило, в якості базису записується якийсь *найпростіший випадок, при якому відповідь виходить відразу, без використання рекурсії*.
- Існує таке поняття як ***крок рекурсії або рекурсивний виклик.***
- У разі, коли рекурсивна функція *викликається для виконання складного завдання* (НЕ базового випадку) виконується деяка кількість рекурсивних викликів або кроків, з метою зведення задачі до більш простий. І так до тих пір поки не отримаємо базове рішення.

Складна рекурсія

- Можлива трохи складніша схема: функція А викликає функцію В, а та в свою чергу викликає А. Це називається складною рекурсією.



- При цьому виявляється, що описана перша процедура повинна викликати ще не описану.

- Щоб це було можливо, потрібно використовувати ***опис функції В до її використання***.

Складна рекурсія. Приклад

- Обчислити значення виразу $\frac{x^n}{n!}$,

```
#include <iostream>
using namespace std;
int pow (int, int);
double calc (int x, int n)
{
   return (double) pow (x, n) / n;        // виклик функції pow
}

int pow (int x, int n)
{
   if (n == 1) return x;
   return x * calc (x, n - 1);              // виклик функції calc
}

int main ()
{
   int n, x;
   cout << "n ="; cin >> n;
   cout << "x ="; cin >> x;
   double a = calc (x, n);                  // виклик рекурсивної функції
   cout << a;
   cin.get (); cin.get ();
   return 0;
}
```

Префіксная і постфіксний форма запису

- Якщо процедура викликає сама себе, то, по суті, це призводить до повторного виконання містяться в ній інструкцій, що аналогічно роботі циклу.

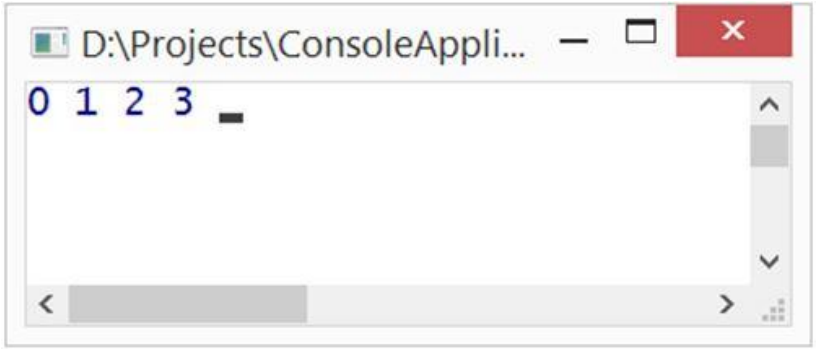
- При цьому розрізняють префіксних і постфіксний форми запису.

Префіксная і постфіксний форма запису

Префіксная форма

Спочатку - рекурсивний виклик,
потім - дії

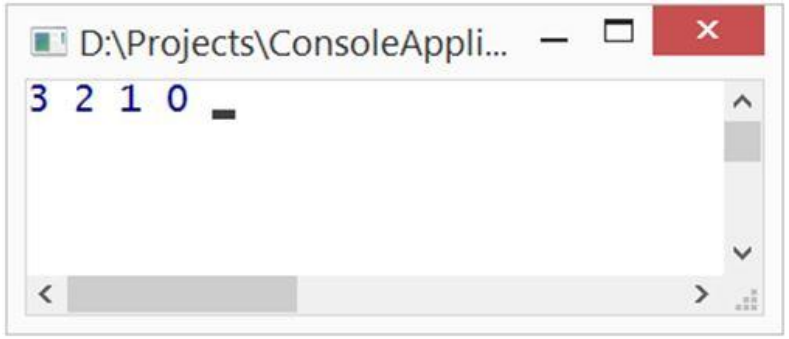
```
...
void func(int num)
{
  if (num > 0) func(num - 1);
  cout << num << " ";
}
...
```



Постфіксна форма

Спочатку - дії,
потім - рекурсивний виклик

```
...
void func(int num)
{
  cout << num << " ";
  if (num > 0) func(num - 1);
}
...
```



Рекурентні співвідношення

- У багатьох випадках в основі рекурсії лежать ***рекурентні співвідношення.***
- **Рекурентне співвідношення** - це співвідношення виду

$$a_n = f(n, a_{n-1}, a_{n-2}, \dots, a_{n-p})$$

- де кожний член послідовності an виражається через p попередніх членів.
- Обчислення необхідного елемента послідовності складатиметься в періодичному оновленні значень цієї послідовності.
- Кожне таке оновлення називається ***ітерацією***, а процес повторення ітерацій - ***ітеруванням***.

Рекурсія або ітерація

- ***Ітерація*** - організація обробки даних, при якій дії повторюються багато разів, *не наводячи при цьому до викликів самих себе* (на відміну від рекурсії).

- Розглянемо обчислення факторіала у вигляді ітераційної і рекурсивної процедури.

Рекурсія або ітерація

Ітераційна функція	***Рекурсивна функція***

#include <iostream>
 	
using namespace std;
 	
int fact(int num)
{
int rez = 1;
for (int i = 1; i <= num; i++)
rez *= i;
return rez;
}
 	
int main()
{
cout << fact(3);
cin.get();
return 0;
}
	#include <iostream>
	
	using namespace std;
	
	int fact(int num)
	{
	if(num==1) return 1;
	return num*fact(num - 1);
	}
	
	int main()
	{
	cout << fact(3);
	cin.get();
	return 0;
	}

Рекурсія або ітерація

- Виклик функції тягне за собою деякі додаткові накладні витрати, пов'язані з передачею керування і аргументів на функцію, а також поверненням обчисленого значення. Тому ітераційна функція обчислення факторіала буде трохи більш швидким рішенням. Найчастіше ***ітераційні рішення працюють швидше рекурсивних.***
- Будь-які рекурсивні процедури і функції, що містять всього один рекурсивний виклик самих себе, ***легко замінюються ітераційними циклами.***
- Ще одним ***недоліком*** рекурсії є те, що їй може ***не вистачати для роботи стека***. При кожному рекурсивном виклику в стеці зберігається адреса повернення і передані аргументи. ***Якщо рекурсивних викликів занадто багато, відведений обсяг стека може бути перевищений. (Наприклад, рекурсивне обчислення факторіала негативного числа).***
- Однак функції, що викликають себе два і більше разів частіше за все не мають простого нерекурсівние аналога. У цьому випадку безліч викликаються процедур ніяк не ланцюжок, а ціле дерево.
- Існують широкі класи задач, коли обчислювальний процес повинен бути організований саме таким чином. Якраз для них рекурсія буде найбільш простим і природним способом вирішення. Крім того, рекурсивні алгоритми, як правило, набагато простіше з логічної точки зору, ніж ітераційні.

Контекст виконання, стек

- ***Інформація про процес виконання запущеної функції зберігається в її контексті виконання*** (execution context).
- **Контекст виконання** - спеціальна внутрішня структура даних, яка містить інформацію про виклик функції. Вона включає в себе конкретне місце в коді, на якому знаходиться інтерпретатор, локальні змінні функції та іншу службову інформацію.
- Один виклик функції має рівно один контекст виконання, пов'язаний з ним.

- Коли функція виконує вкладений виклик, відбувається наступне:
  1. Виконання поточної функції припиняється.
  2. Контекст виконання, пов'язаний з нею, запам'ятовується в спеціальній структурі даних - стеці контекстів виконання.
  3. Виконуються вкладені виклики, для кожного з яких створюється свій контекст виконання.
  4. Після їх завершення старий контекст дістається з стека, і виконання зовнішньої функції поновлюється з того місця, де вона була зупинена.

Контекст виконання, стек

- Розглянемо приклад

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

int  pow (int x,int n)
{
 if (n == 1)
     {   return x;  }
 else
     {   return x * pow (x, n - 1); }
}
int main ()
{
 int tt=pow (2, 3));  // 8
 cout << tt;
 cin.get ();
 return 0;
}
```

Коли функція pow (x, n) викликається, виконання ділиться на дві гілки:

```
          if n == 1 = x
         /
pow (x, n) =
         \
          else = x * pow (x, n - 1)
```

Контекст виконання, стек

Коли функція pow (x, n) викликається, виконання ділиться на дві гілки:

```
             if n == 1 = x
            /
pow (x, n) =
            \
             else = x * pow (x, n - 1)
```

- 1. Якщо n == 1, тоді все просто. Ця гілка називається базою рекурсії, тому що відразу ж призводить до результату: pow (x, 1) однт x.
- 2. Ми можемо виразити pow (x, n) у вигляді: x * pow (x, n - 1). Що в математиці записується як: xn = x * xn-1. Ця гілка - крок рекурсії: *ми зводимо задачу до більш простої дії* (множення на x) і *більш простої аналогічної задачі* (pow з меншим n). Наступні кроки спрощують задачу все більше і більше, поки n не досягає 1.
- Функція pow рекурсивно викликає саму себе до n == 1.

Контекст виконання, стек

- Наприклад, рекурсивний варіант обчислення pow (2, 4) складається з кроків:
- 1. pow (2, 4) = 2 * pow (2, 3)
- 2. pow (2, 3) = 2 * pow (2, 2)
- 3. pow (2, 2) = 2 * pow (2, 1)
- 4. pow (2, 1) = 2

```
int  pow (int x,int n)
{
 if (n == 1)
      {   return x;  }
 else
      {   return x * pow (x, n - 1); }
}
```

Контекст виконання, стек

- Глибина рекурсії в даному випадку склала **3** - максимальне число контекстів, що буде одночасно збережено в стеці.
- *Звернемо увагу на вимоги до пам'яті.* Рекурсія призводить **до зберігання всіх даних для незакінчених зовнішніх викликів в стеці, і в даному випадку це призводить до того, що зведення в ступінь n зберігає в пам'яті n різних контекстів.**

Контекст виконання, стек

- Реалізація зведення в степінь через цикл набагато ***більш економна***:

```
function pow (x, n)
 {
  let result = 1;
  for (let i = 0; i <n; i ++)
     {
        result * = x;
      }
  return result;
 }
```

- Ітеративний варіант функції pow використовує один контекст, в якому будуть послідовно змінюватися значення i і result. При цьому обсяг витрачається пам'яті невеликий, фіксований і не залежить від n.

Контекст виконання, стек

- *Будь-яка рекурсія може бути перероблена в цикл. Як правило, варіант з циклом буде ефективніше.*

- Але переробка рекурсії в цикл може бути нетривіальною, особливо коли в функції в залежності від умов використовуються різні рекурсивні підвизови, результати яких об'єднуються, або коли розгалуження більш складне. **Оптимізація може бути непотрібною і абсолютно не вартою зусиль**.

- Часто код з використанням рекурсії більш короткий, легкий для розуміння і підтримки. *Оптимізація потрібно не скрізь, як правило, нам важливий хороший код, тому вона і використовується*.

Приклад 1

- *Обчислити суму чисел в інтервалі, заданому числами, що вводяться з клавіатури. Використати рекурсивную функцію.*
- **Рішення.** Умовою закінчення рекурсії стане ситуація, коли верхня межа на одиницю більше нижньої межі, тобто інтервал заданий двома сусідніми цілими числами.

```
#include <iostream>
using namespace std;
int sum (int y, int x);

int main ()
{
int a, b;
cout << "Enter 1-st number:" << endl; cin >> a;
cout << "Enter 2-st number:" << endl; cin >> b;
cout << sum (b, a) << endl;
return 0;
}
```

```
int sum (int y, int x)
{
int s = 0;
if ((y - 1) == x)
  s = y + x;
else
  s = y + sum (y - 1, x);
return s;
}
```

Приклад 2

- *Звести число в ступінь. Використати рекурсивную функцію.*
- Рішення. Тут умовою закінчення рекурсії буде рівність нулю числа-ступеня. Обов'язково слід передбачити виклики функції для парного і непарної ступеня.

```
#include <iostream>
using namespace std;
int power (long int x, unsigned int y);

int main ()
{
int a, b;
cout << "Enter number:" << endl; cin >> a;
cout << "Enter power:" << endl; cin >> b;
cout << power (a, b) << endl;
return 0;
}
```

```
int power (long int x, unsigned int y)
{
  int d = 0;
  if (y == 0)
      d = 1;
  else if (y == 1)
      d = x;
      else if (y% 2 == 0)
          d = power (x * x, y / 2);
      else
          d = x * power (x * x, y / 2);
  return d;
}
```

Дякую за увагу!