Лекція №24

Визначення та види рекурсії.

Рекурсивна функція (або просто ***"рекурсія"***) в мові C++ — це функція, яка викликає саму себе. Наприклад:

```
1  #include <iostream>
2
3  void countOut(int count)
4  {
5  std::cout << "push " << count << '\n';
6  countOut(count-1); // функція countOut() рекурсивно викликає саму себе
7  }
8
9  int main()
10 {
11 countOut(4);
12
13 return 0;
14 }
```

При виклику функції countOut(4) на екран виведеться push 4, а потім буде виклик countOut(3). countOut(3) виведе push 3 і викличе countOut(2). Послідовність виклику countOut(n) інших функцій countOut(n-1) повторюється нескінченну кількість разів (аналог нескінченного циклу). Спробуйте запустити у себе.

При кожному виклику функції, певні дані поміщаються в стек викликів. Оскільки функція countOut() ніколи нічого не повертає (вона просто знову викликає countOut()), то дані цієї функції ніколи не витягуються зі стеку! Отже, в якийсь момент пам'ять стеку закінчиться і відбудеться **переповнення стеку**.

Умова завершення рекурсії

Рекурсивні виклики функцій працюють точно так же, як і звичайні виклики функцій. Однак, програма, наведена вище, ілюструє найбільш важливу відмінність простих функцій від рекурсивних: ви повинні вказати умову завершення рекурсії, в протилежному випадку — функція виконуватиметься нескінченну кількість разів (фактично до тих пір, поки не закінчиться пам'ять в стеці викликів).

Умова завершення рекурсії — це умова, при якій рекурсивна функція перестане викликати саму себе. В цій умові зазвичай використовується **оператор if**.

Ось приклад функції, наведеної вище, але вже з умовою завершення рекурсії (і ще з одним додатковим виводом тексту на екран):

```
 1  #include <iostream>
 2
 3  void countOut(int count)
 4  {
 5      std::cout << "push " << count << '\n';
 6
 7      if (count > 1) // умова завершення
 8          countOut(count-1);
 9
10      std::cout << "pop " << count << '\n';
11 }
12
13 int main()
14 {
15     countOut(4);
16     return 0;
17 }
```

Коли ми запустимо цю програму, то countOut() почне виводити:

```
push 4
push 3
push 2
push 1
```

Якщо зараз подивитися на стек викликів, то побачимо наступне:

```
countOut(1)
countOut(2)
countOut(3)
countOut(4)
main()
```

Через умову завершення, countOut(1) не викличе countOut(0): умова if не виконається, і виведеться pop 1 і countOut(1) завершить своє виконання. На цьому етапі countOut(1) витягується зі стеку, і керування повертається до countOut(2).

countOut(2) відновлює виконання в точці після виклику countOut(1), і тому виведеться pop 2, а потім countOut(2) завершиться. Рекурсивні виклики функцій countOut() поступово витягуються зі стеку до тих пір, поки не будуть видалені всі екземпляри countOut().

Таким чином, результат виконання програми, наведеної вище:

```
push 4
push 3
push 2
push 1
pop 1
pop 2
pop 3
pop 4
```

Варто відзначити, що push виводиться в порядку спадання, а pop — в порядку зростання. Справа в тому, що push виводиться до виклику рекурсивної функції, а pop виконується (виводиться) після виклику рекурсивної функції, коли всі екземпляри countOut() витягуються зі стеку (це відбувається в порядку, зворотному тому, в якому ці екземпляри були додані в стек).

Тепер, коли ми поговорили про основний механізм виклику рекурсивних функцій, давайте поглянемо на інший тип рекурсії, який більш поширений:

```
 1  // Повертаємо суму всіх чисел між 1 і value
 2  int sumCount(int value)
 3  {
 4      if (value <= 0)
 5          return 0; // базовий випадок (умова завершення)
 6      else if (value == 1)
 7          return 1; // базовий випадок (умова завершення)
 8      else
 9          return sumCount(value - 1) + value; // рекурсивний виклик функції
10  }
```

Виявити рекурсію з першого погляду на код не так вже й легко. Кращим варіантом буде подивитися, що станеться при виклику рекурсивної функції з певним значенням. Наприклад, подивимося, що станеться при виклику вищенаведеної функції з value = 4:

sumCount(4). 4 > 1, тому повертається sumCount(3) + 4
sumCount(3). 3 > 1, тому повертається sumCount(2) + 3

sumCount(2). 2 > 1, тому повертається sumCount(1) + 2
sumCount(1). 1 = 1, тому повертається 1. Це умова завершення рекурсії

Тепер подивимося на стек викликів:

sumCount(1) повертає 1
sumCount(2) повертає sumCount(1) + 2, тобто 1 + 2 = 3
sumCount(3) повертає sumCount(2) + 3, тобто 3 + 3 = 6
sumCount(4) повертає sumCount(3) + 4, тобто 6 + 4 = 10

На цьому етапі вже легше побачити, що ми просто додаємо числа між 1 і значенням, яке надав caller. На практиці рекомендується вказувати **коментарі** біля рекурсивних функцій, щоб полегшити життя не тільки собі, але, можливо, і іншим людям, які переглядатимуть ваш код.

Рекурсивні алгоритми

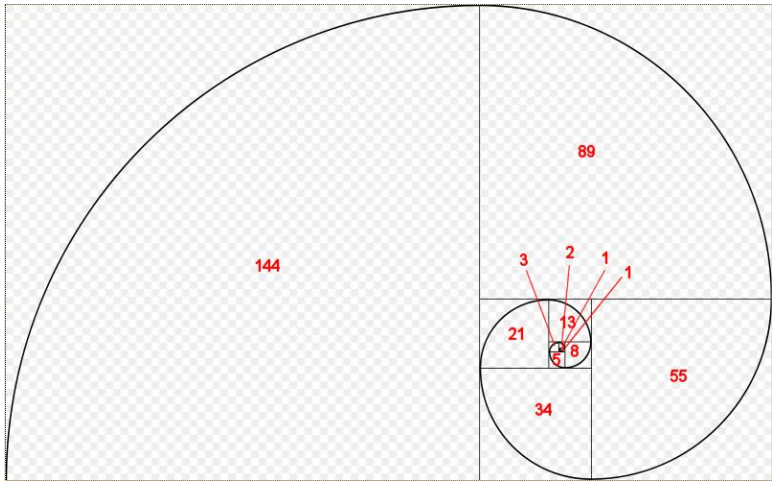
Рекурсивні функції зазвичай вирішують проблему, спочатку знайшовши рішення для підмножин проблеми (рекурсивно), а потім модифікуючи це «підрішення», щоб дістатися вже до вірного рішення. У вищенаведеному прикладі, алгоритм sumCount(value) спочатку вирішує sumCount(value-1), а потім додає значення value, щоб знайти рішення для sumCount(value).

У багатьох рекурсивних алгоритмах деякі дані вводу видають передбачувані дані виводу. Наприклад, sumCount(1) має передбачуваний вивід 1 (ви можете легко це обчислити і перевірити самостійно). Випадок, коли алгоритм при певних даних вводу видає передбачувані дані виводу, називається **базовим випадком**. Базові випадки виконуються як умови для завершення виконання алгоритму. Їх часто можна ідентифікувати, розглядаючи результати виводу для наступних значень вводу: 0, 1, «» або null.

Числа Фібоначчі

Одним з найбільш відомих математичних рекурсивних алгоритмів є **послідовність Фібоначчі**. Послідовність Фібоначчі можна побачити навіть в природі: розгалуження дерев, спіраль мушлі, плоди ананасу тощо.

Спіраль Фібоначчі виглядає наступним чином:



Кожне з чисел Фібоначчі — це довжина горизонтальної сторони квадрата, в якій знаходиться дане число. Математично числа Фібоначчі визначаються наступним чином:

F(n) = 0, якщо n = 0
1, якщо n = 1
f(n-1) + f(n-2), якщо n > 1

Отже, досить просто написати рекурсивну функцію для обчислення n-го числа Фібоначчі:

```
 1  #include <iostream>
 2
 3  int fibonacci(int number)
 4  {
 5      if (number == 0)
 6          return 0; // базовий випадок (умова завершення)
 7      if (number == 1)
 8          return 1; // базовий випадок (умова завершення)
 9      return fibonacci(number-1) + fibonacci(number-2);
10 }
11
12 // Виводимо перші 13 чисел Фібоначчі
13 int main()
14 {
15     for (int count=0; count < 13; ++count)
16         std:: cout << fibonacci(count) << " ";
17
18     return 0;
19 }
```

Результат виконання програми:

0 1 1 2 3 5 8 13 21 34 55 89 144

Помітили? Це ті ж числа, що і в спіралі Фібоначчі.

Рекурсія vs. Ітерації

Найбільш популярне питання, яке задають про рекурсивні функції: «Навіщо використовувати рекурсивну функцію, якщо завдання можна виконати і за допомогою ітерацій (використовуючи **цикл for** чи **цикл while**)?». Виявляється, ви завжди можете вирішити рекурсивну проблему ітеративно. Однак, для нетривіальних випадків, рекурсивна версія часто буває набагато простіша як для написання, так і для читання. Наприклад, функцію обчислення n-го числа Фібоначчі можна написати і за допомогою ітерацій, але це складніше!

Ітеративні функції (ті, які використовують цикли for або while) майже завжди більш ефективні, ніж їх рекурсивні аналоги. Це пов'язано з тим, що кожен раз, при виконанні функції, витрачається певна кількість ресурсів на додання і витягування фреймів зі стеку. Ітеративні функції витрачають набагато менше цих ресурсів.

Це не означає, що ітеративні функції завжди є кращим варіантом. Іноді рекурсивна реалізація може бути чистішою і простішою, а деякі додаткові витрати можуть бути більш ніж виправдані, звівши до мінімуму труднощі при майбутній підтримці коду, особливо, якщо алгоритм не вимагає занадто багато часу для пошуку рішення.

Загалом, рекурсія є хорошим вибором, якщо виконується більшість з наступних тверджень:

рекурсивний код набагато простіше реалізувати;

глибина рекурсії може бути обмежена;

ітеративна версія алгоритму вимагає управління стеком даних;

це не критична частина коду, яка напряму впливає на продуктивність програми.