РОЗДІЛ 7
РЕКУРСІЯ

Одним зі способів опису об'єктів є рекурсія. Рекурсивними можуть бути правила, що описують структуру виразів деякої мови, означення математичних функцій, алгоритми тощо. Рекурсія є одним із фундаментальних понять програмування й математики. Завдяки їй різноманітним об'єктам можна дати зрозумілий і компактний опис.

У цьому розділі розглянуто рекурсивні функції мови C++. Проте спочатку, щоб краще розуміти їх виконання, розглянемо, як узагалі виконуються виклики функцій.

7.1. Виконання виклику функції

7.1.1. Пам'ять виклику функції

Сукупність змінних, що утворюються під час виклику функції (підпрограми), має назву **пам'ять виклику функції**, або, не зовсім точно, – **локальна пам'ять** функції. Змінні в цій пам'яті називаються **локальними** й відповідають **параметрам** та **іменам змінних**, означеним у тілі функції.

Локальна пам'ять функції містить ще один елемент – посилання на місце, з якого має виконуватися програма після закінчення виклику. Наприклад, головна функція на с. 99, що містить три виклики функції `swap`, після цих викликів продовжується трьома різними інструкціями. Місце продовження називається

точкою повернення з функції, а посилання на неї зберігається під час виконання виклику функції.

7.1.2. Огляд процесу виконання виклику

1. Виділяється пам'ять для точки повернення й параметрів функції. Посилання на точку повернення з функції запам'ятовується.
2. Обчислюються значення аргументів для параметрів-значень, посилання на пам'ять аргументів для параметрів-посилань. Відбувається підстановка аргументів.
3. Виділяється пам'ять, відповідна до локальних імен змінних (окрім локальних статичних змінних (див. підрозд. 8.2)).
4. Виконуються інструкції тіла функції до інструкції повернення.
5. Якщо підпрограма не є **`void`**-функцією, то значення, що повертається з її виклику, копіюється у пам'ять функції, яка містила виклик.
6. Функція, що містила виклик, продовжується з точки повернення.

Змінні в локальній пам'яті функції не відповідають іменам у функції, що містила виклик, тобто ця пам'ять **недоступна** після того, як виклик закінчено. Вона **вважається звільненою**; її можна використовувати для наступного виклику цієї або іншої функції.

Відбувається *логічне звільнення*, тобто зміст локальної пам'яті не змінюється, але стає недоступним.

7.1.3. Автоматична пам'ять, або програмний стек

Ділянки пам'яті викликів функцій утворюються та звільняються в спеціальній області пам'яті процесу виконання програми – **автоматичній пам'яті**. Називається вона так тому, що за виконання викликів функцій пам'ять виділяється та звільняється без явних вказівок у програмі, написаній мовою високого рівня, тобто автоматично.

Під час виконання викликів функцій ділянки автоматичної пам'яті виділяються та звільняються за принципом "останньою зайнято – першою звільнено". Якщо складати аркуші паперу в стос і брати їх тільки згори, то аркуш, що потрапив до стосу останнім, забирають першим. Англійською стос називається *stack* (стек), а кладуть і беруть аркуші за принципом "*Last In – First Out*" (LIFO), тобто "останнім прийшов – першим пішов". Тому автоматичну пам'ять програми також називають **програмним стеком**.

Аналогічно набій, заштовхнутий у магазин автомата останнім, вилітає першим, а заштовхнутий першим (він на дні магазина) вилітає останнім. Можна вистрілити набій з магазина й на його місце додати новий. Так само, коли послідовно виконуються два виклики функцій у тілі функції, то для другого виклику виділяється пам'ять, звільнена після першого.

Приклад. Розглянемо імітування такої програми:

```
#include <iostream>
using namespace std;
int f(int x)
{ return ++x; }
int g(int& x)
{ return x/=2;  }
int main(){
  int a=12;
  cout << f(g(a)) << ' ';
  cout << a << endl;
  system("pause"); return 0;
}
```

prog015.cpp

Імітуючи програму, до імен змінних, оголошених у функціях, приписуємо імена цих функцій, наприклад **`main.a`** або **`f.x`**. Ім'я параметра-посилання та ім'я змінної, що є відповідним аргументом у виклику, позначають одну й ту саму змінну. Через **`f.r`** та **`g.r`** позначимо значення, що повертаються з функцій.

Що виконується	Змінні всіх функцій		
	`main.a`		
`int a=12`	12		
початок `out<<f(g(a))`	12		
виклик `f(g(a))`	12	`f.x`	
початок `f.x=g(a)`	12	?	
виклик `g(a)`	`g.x`	?	
`g.x/=2`	6	?	
`g.r=g.x`	6	?	
повернення з `g`	6	?	`g.r`=6
закінчення `f.x=g(a)`	6	6	
`++f.x`	6	7	
`f.r=f.x`	6	7	
повернення з `f`	6	7	`f.r`=7
виведення `f(g(a)): 7`	6		
виведення `a: 6`	6		

Виконання виклику `f(g(a))` починається з обчислення виразу `g(a)` – аргументу для параметра `f.x`, тому починається виклик функції `g` з аргументом `a`. Цей аргумент підставляється за посиланням, тому імена `g.x` і `main.a` позначають одну й ту саму змінну. З виклику функції `g` повертається `6` і присвоюється параметру `f.x`. Пам'ять функції `g` звільняється. Тільки тепер виконуються інструкції функції `f`. Після їх закінчення в головну функцію повертається значення `7`. Пам'ять виклику функції `f` звільняється. У головній функції виводиться значення `7`, отримане з виклику, а потім – значення змінної `main.a`. ◂

7.2. Ознайомлення з рекурсією

7.2.1. Поняття та приклади рекурсії

Означення називається **рекурсивним**, якщо воно задає елементи певної множини за допомогою інших елементів цієї самої множини. Об'єкт, заданий рекурсивним

означенням, також називається **рекурсивним**, а використання таких означень – **рекурсією**.

Приклади

1. Значення функції "факторіал" можна задати початковим елементом $0! = 1$ і рекурентним співвідношенням $n! = n \cdot (n\text{-}1)!$. Усі елементи цієї множини, крім першого, означаються рекурсивно. Узагалі, будь-яке рекурентне співвідношення разом із початковими умовами є прикладом рекурсивного означення.

2. Арифметичні вирази з константами, знаком операції + і дужками опишемо так:

а) константа є арифметичним виразом;

б) якщо E та F – арифметичні вирази, то $(E)+(F)$ – також вираз;

в) інших арифметичних виразів, крім утворених за пп. а) та б), не існує.

Описаними виразами є, наприклад, 1, 2, (1)+(2), ((1)+(2))+(1).

3. Інструкції мови C++, що задають розгалуження й цикли, самі містять інструкції. Отже, представники поняття "інструкція" є рекурсивними. ◂

Рекурсивне означення повинно не мати "зачарованого кола", коли в означенні об'єкта використовується він сам або інші об'єкти, задані за його допомогою.

Приклад. Змінимо означення функції "факторіал": $n! = n \cdot (n\text{-}1)!$ за $n > 0$, $0! = 1!$. Значення функції від 1 виражається через її ж значення від 0, яке, у свою чергу, – через значення від 1. За цим "означенням" не можна дізнатися, яким числом є 1!. ◂

7.2.2. Приклади рекурсивних функцій

Функції, що містять виклики самих себе, як і самі ці виклики, називаються **рекурсивними**. Виклик рекурсивної функції виконується так само, як і виклик будь-якої функції.

Приклади

1. На клавіатурі набираються цілі числа, не рівні нулю. Поява 0 означає кінець уведення. Задача: прочитати числа й видати їх у зворотному порядку (кінцевий 0 не виводити).

Перше число треба вивести останнім, друге – передостаннім і т. д. Отже, для обробки входу потрібно прочитати перше число

і, якщо це не 0, то *в такий самий спосіб* обробити решту входу й потім вивести перше число. Якщо прочитано 0, то обробку вхідних даних закінчено. Ці дії описує рекурсивна функція **outReverse.**

```
#include <iostream>
using namespace std;
void outReverse()
{ int n;
  cin >> n;
  if (n==0) return;  // уведено 0 – повернення
  outReverse();      // уведено не 0 – заглиблення
  // після повернення з рекурсивного виклику
  cout << n << " ";
  return;
}
int main(){
  cout << "Enter integers, the last should be 0\n";
  outReverse();
  cout<<endl;
  system("pause"); return 0;
}
```

prog016.cpp

Уведене число записується в *локальну* змінну **n** функції; кожен виклик додає до програмного стека новий екземпляр **n**. Після введення 0 виклики закінчуються в порядку, зворотному до того, в якому починалися. Звідси й значення локальних змінних **n** виводяться у зворотному порядку. У головній функції потрібен лише виклик функції **outReverse**.

2. Прочитати невід'ємне число *n* типу **int** (як звичайне десяткове) та основу системи числення *p*, $2 \le p < 37$. Вивести *p*-ковий запис числа. Якщо $p > 10$, то для зображення чисел 10, 11, 12, …, скористатися "цифрами" **A, B, C, ….**

Значення молодшої цифри в *p*-ковому записі числа є остачею від ділення числа на *p*. Далі, узявши замість числа частку від його ділення на *p*, *так само* отримаємо значення наступної цифри. Так можна діяти, доки не залишиться число менше *p*. Проте цифру, *отриману першою*, треба *вивести останньою*, тому запам'ятаємо значення молодшої цифри, далі рекурсивно виведемо старші цифри, а потім – молодшу цифру.

Головна функція вводить два числа – число *n* та основу системи числення *p*, за їх коректності викликає функцію **outNP** виведення *p*-кових цифр числа *n*, інакше повідомляє про помилку.

У функції виведення цифр **outNP(n, p)** заглиблюємося в рекурсію, якщо $n \geq p$ (число n має не менше двох p-кових цифр), і виводимо останню цифру. Можливим значенням цифр від 0 до 35 відповідають цифри – символи від **'0'** до **'9'** та від **'A'** до **'Z'**. Для перетворення цифри на її символьне зображення використовується допоміжна функція **digit**.

```
#include <iostream>
using namespace std;
char digit(int v) // цифра за значенням v
//pre: 0<=v<=35
{ if(v<10) return char(int('0')+v);
  else return char(int('A')+v-10);
}
void outNP(int n, int p)
// виведення p-кових цифр числа n
//pre: n>=0 && p>=2
{ if(n>=p) outNP(n/p, p); // заглиблення
  cout << digit(n%p);  // виводиться остання цифра
  return;
}
int main() {
  int n, p;
  cout << "Enter one nonnegative integer and\n" <<
          " one integer between 2 and 36 >";
  cin >> n >> p;
  if (n>=0 && 2<=p && p<=36) outNP(n,p);
  else cout<<"You enter wrong numbers";
  cout<<endl; system("pause"); return 0;
}
```

prog017.cpp

Для перевірки програми задайте число 2147483647 (найбільше число типу **int**) та основи 2, 8, 16, 32. Відповідними результатами мають бути **11…1** (31 "**1**"), **17777777777**, **7FFFFFF**, **1VVVVVV**. Перевірте також кілька чисел із максимальною основою 36, наприклад 35, 71 та 1295 (результатами мають бути **Z**, **1Z** та **ZZ**). Також слід перевірити некоректні числа й основи системи числення. ◀

З кожним рекурсивним викликом зайнята частина програмного стека збільшується, а із закінченням виклику – зменшується. Розмір стека обмежений, тому можлива ситуація (особливо за виконання рекурсивних функцій), коли пам'яті в стеку забракне й програма завершиться аварійно.

У рекурсивній функції обов'язково має бути *умова*, за істинності якої відбувається повернення з виклику (див. приклади вище). Ця умова визначає *дно рекурсії*, яке під час виконання функції *обов'язково має досягатися*, інакше виклики призведуть до переповнення програмного стека або інших непередбачуваних наслідків.

Приклади

1. Для функції **outNP** із попередньої програми кожен виклик зменшує значення параметра **n**, тому дно рекурсії в ній досягається.

2. Розглянемо функцію з *невдалою умовою* повернення з рекурсії.

```
void badFunc(int x)
{ if(x==2) {cout << x; return;} // повертаємося
  badFunc(x-2);                 // заглиблюємося
}
```

За виконання виклику **badFunc(6)** відбуваються рекурсивні виклики з аргументами **4** та **2**. За **x==2** заглиблення в рекурсію немає, тому виводиться **2**, а потім послідовно закінчуються виклики з аргументами **2, 4** та **6**. Проте виклик **badFunc(5)** приведе до рекурсивних викликів з аргументами **3, 1, –1, –3, ...,** в яких умова повернення **x==2** ніколи не стане істинною, тому виконання рекурсивних викликів *переповнить програмний стек*. ◀

У цьому розділі розглядається тільки *пряма рекурсія*, коли функція містить виклики самої себе. У програмуванні також використовується *непряма рекурсія*, коли функції містять взаємні виклики.

Вправа 7.1. Що буде виведено на екран, якщо під час виконання програми ввести а) 10; б) 7; в) 5?

```
#include <iostream>
using namespace std;
void func (int n){
  if (n>7) func(n-1);
  cout<<n<<" ";
  return;
}
int main()
{ int n;
  cout <<"Enter one integer\n"; cin >> n;
  func(n);
  cout<<endl; system("pause");return 0;
}
```

7.3. "Підводні камені" рекурсії

7.3.1. Ханойські вежі

На дошці – три стрижні: 1, 2, 3. На першому розміщено вежу з n дисків; нижній диск має найбільший діаметр, а діаметр кожного наступного менший за діаметр попереднього. За один хід із будь-якого стрижня можна взяти верхній диск і перемістити на інший стрижень, але дозволено класти диск лише на дошку або на диск більшого діаметра. Треба перемістити всю вежу зі стрижня 1 на стрижень 3.

Ця гра називається "Ханойські вежі". За легендою цю гру почали понад тисячу років тому ченці в одному монастирі поблизу Ханоя у В'єтнамі. У ченців було $n = 64$ диски. Коли вони закінчать гру, настане кінець Усесвіту. Розв'язком цієї гри-задачі є послідовність перенесень дисків. Напишемо програму виведення позначень цих перенесень.

Щоб перенести вежу з n дисків зі стрижня 1 на стрижень 3, необхідно перенести вежу з n-1 диска на стрижень 2, потім перенести нижній диск на стрижень 3, потім – вежу з 2 на 3. При перенесенні вежі з 1 на 2 допоміжним буде стрижень 3, а при перенесенні з 2 на 3 – 1, причому жодна інша послідовність дій неможлива. Отже, розв'язання задачі для вежі висотою n описано за допомогою розв'язання для вежі висотою n-1, тобто *рекурсивно*.

Нехай **diskMove(a,b)** позначає перенесення одного диска зі стрижня **a** на стрижень **b**, а **tower(h,a,b,c)** – перенесення вежі висотою **h** з **a** на **b** з використанням стрижня **c** як допоміжного. За **h** > 1 виконання **tower(h,a,b,c)** буде таким:

```
tower(h-1,a,c,b);
diskMove(a,b);
tower(h-1,c,b,a)
```

За **h** = 1 переносять тільки один диск, тобто виконують **diskMove(a,b)**. Отже, очевидною є така програма:

```
#include <iostream>
using namespace std;
void diskMove(int diskFrom, int diskTo)
{ cout << diskFrom << "->" << diskTo << ' ';
  return;
}
```

```
void tower(int height, int from,int to, int via)
//height - висота, from - з, to - на, via - через
{ if(height==1) diskMove(from,to);
  else
  { tower(height-1,from,via,to);
    diskMove(from,to);
    tower(height-1,via,to,from);
  }
}
int main()
{ int n;
  cout << "Hanoi towers\n";
  cout << "Enter a number of disks>";
  cin >> n; if (n<1) n=1;
  tower(n,1,3,2);
  cout<<endl;
  system("pause");return 0;
}
```

prog018.cpp

Визначимо кількість перенесень дисків у вигляді функції $s(n)$, де n – висота вежі. Очевидно, що $s(1)=1$ і $s(n)=2\cdot s(n\text{-}1)+1$. Як бачимо, $s(2)=3$, $s(3)=7$, $s(4)=15$ тощо. Кожне наступне значення подвоюється з додаванням 1, тому неважко переконатися, що $s(n)=2^n\text{-}1$. Значення $s(64)$ приблизно дорівнює 10^{19}. Якщо припустити, що ченці переносять один диск щосекунди, то для перенесення такої вежі потрібно більше 10^{12} років!

Рекурсивна функція без жодного циклу може легко приховати величезні обсяги обчислень. Як і будь-який потужний засіб, рекурсія вимагає обережного використання.

7.3.2. Глибина рекурсії й загальна кількість рекурсивних викликів

З рекурсивними функціями пов'язано два важливих поняття – глибина рекурсії й загальна кількість викликів, породжених викликом рекурсивної підпрограми. Будемо відрізняти глибину рекурсії, на якій перебуває виклик, і глибину рекурсії, породжену викликом.

Глибина рекурсії, на якій перебуває виклик підпрограми – це кількість рекурсивних викликів, розпочатих і не закінчених у момент початку цього виклику.

Глибина рекурсії, породжена викликом – це максимальна кількість рекурсивних викликів, які розпочато й не закінчено після початку цього виклику.

Наприклад, у програмі "Ханойські вежі" виклик функції **tower** із висотою h перебуває на глибині рекурсії $n - h$ і породжує глибину $h - 1$.

Коли виконується виклик функції, який перебуває на глибині рекурсії m, одночасно існує $m+1$ екземпляр локальної пам'яті функції. Кожен екземпляр займає ділянку певного розміру, тому збільшення глибини, як було сказано вище, може призвести до *переповнення програмного стека*.

Загальна кількість рекурсивних викликів, породжених викликом рекурсивної функції, – це кількість викликів, виконаних між його початком і завершенням.

Наприклад, у задачі "Ханойські вежі" кожен виклик задає одне перенесення диска та, якщо він не є найбільш заглибленим, породжує два рекурсивних виклики. Нам уже відомо, що перенесення вежі висотою n вимагає 2^n-1 перенесень дисків, тому між початком і закінченням виклику з аргументом n виконуються 2^n-2 рекурсивні виклики.

Загальна кількість породжених викликів визначає тривалість виконання виклику. Як бачимо, кількість викликів у задачі "Ханойські вежі" за значень n уже порядку кількох десятків призводить до неприпустимо довгого виконання програми.

Записуючи рекурсивну функцію, необхідно вміти оцінити можливу глибину рекурсії, розмір пам'яті виклику функції й загальну кількість рекурсивних викликів.

7.3.3. Приклад недоречного використання рекурсії

Повернемося до чисел Фібоначчі, що визначаються рекурентним співвідношенням $F_n = F_{n-1}+F_{n-2}$, $n \geq 2$, і початковими умовами $F_0 = 0$, $F_1 = 1$. За цим означенням дуже легко написати функцію обчислення числа Фібоначчі за його номером.

```
int f(int n) //pre: n>=0, інакше повертаємо -1
{  if (n<0) return -1;
  if (n==0) return 0;
  if (n==1) return 1;
  return f(n-2)+f(n-1);      // два виклики
}
```

ЦЕ ДУЖЕ ПОГАНА ФУНКЦІЯ!!! Її головний недолік: вона *породжує абсолютно непотрібні багаторазові обчислення тих самих величин*. Розглянемо, наприклад, обчислення `f(6)`. Очевидно, що виклик `f(4)` виконується двічі, `f(3)` – тричі, `f(2)` – п'ять разів, `f(1)` – вісім разів, `f(0)` – п'ять разів.

У загальному ж випадку за $n > 3$ обчислення F_n породжує два виклики `f(n-2)`, три виклики `f(n-3)`, п'ять викликів `f(n-4)`, …, F_{n-1} викликів `f(2)`, F_n викликів `f(1)` і F_{n-1} викликів `f(0)`. Щоб виконати всі ці дії, потрібно часу не менше ніж для перенесення ханойської вежі висотою n. Однак на відміну від задачі про вежі, ці дії не є обов'язковими, адже циклічний алгоритм на с. 116 (п. 6.3.2) вимагає лише одного додавання та двох присвоювань на кожне з чисел $F_2, F_3, \ldots, F_n$ (плюс присвоювання, додавання й порівняння номерів).

Вправа 7.2. Напишіть функцію, що за цілим значенням `n` повертає кількість вкладених викликів наведеної функції `f` за виклику `f(n)`.

Контрольні запитання

7.1. Чому пам'ять, утворену локальними змінними викликів функцій, називають програмним стеком?
7.2. За якої умови у функції можна використовувати імена, не оголошені в ній?
7.3. Яке ім'я називається у функції локальним, а яке – глобальним?
7.4. Чи можна використовувати локальні імена функції в інших функціях, записаних після неї?
7.5. Чи є глобальними змінні функції `main`?

Задача

7.1. Арифметичні вирази з константами, знаками операцій (`+`, `-`, `*` або `/`) і дужками опишемо так:
а) дійсна константа є арифметичним виразом;
б) якщо E та F – арифметичні вирази, то (E+F), (E-F), (E*F), (E/F) – також вирази;
в) інших виразів, окрім утворених за правилами пп. а) та б), не існує.
Написати програму, що за введеним із клавіатури арифметичним виразом виводить його значення.