РОЗДІЛ 6
Циклічні алгоритми

6.1. Прості інструкції повторення обчислень

6.1.1. Циклічне обчислення факторіала

Функція факторіала $n!$ невід'ємного цілого числа n визначається формулою $n! = (n\text{-}1)!\cdot n$ за $n > 0$, $0! = 1$. Звідси $n! = 1\cdot 2\cdot\ldots\cdot(n\text{-}1)\cdot n$. Щоб обчислити цей добуток за $n \geq 0$, можна почати з добутку 1 і далі послідовно отримувати добутки $1\cdot 1$, $1\cdot 1\cdot 2$, $1\cdot 1\cdot 2\cdot 3$, …, $1\cdot 1\cdot 2\cdot\ldots\cdot(n - 1)\cdot n$, щоразу збільшуючи наступний множник.

У цих міркуваннях присутні поняття *добуток* і *наступний множник*. Зобразимо їх цілими змінними **fact** і **k**. За означенням за **k** = 0 маємо **fact** = 1. Візьмемо наступне значення **k**=1. Отже, починаємо зі значень **fact=1**, **k=1**, а далі на кожному кроці множимо **fact** на **k**, зберігаємо добуток у **fact** і збільшуємо **k** на 1. Кроки повторюємо, поки **k** ≤ n.

```
fact=1; k=1;
поки (k<=n) повторювати { fact*=k; k=k+1;}
```

Мовою C++ це виглядає так:

```
fact=1; k=1;              // початок обчислень
while (k<=n)              // повторення кроків
  { fact*=k; k=k+1; }
// після закінчення повторень k=n+1, fact=n!
```

6.1.2. Інструкція циклу з передумовою

У наведеному прикладі циклічні, тобто повторювані, обчислення задано за допомогою **інструкції циклу з передумовою** (**while**-інструкції). Вона має загальний вигляд

while (умова) ***інструкція***

Слово **while** є зарезервованим, дужки обов'язкові, **while (*умова*)** – це **заголовок циклу**, а ***інструкція*** – **тіло**.

Інструкція циклу виконується так. Спочатку обчислюється умова в заголовку. Якщо вона істинна, то виконується тіло циклу та знов обчислюється умова. Якщо вона істинна, то все повторюється. Виконання інструкції циклу закінчується, коли обчислено значення умови **false**, тобто хибність. Отже, в останньому циклі тільки обчислюється умова, а тіло не виконується. Якщо при першому обчисленні умова хибна, то тіло циклу не виконується жодного разу. Перевірку умови циклу й виконання після неї тіла циклу інколи називають **ітерацією циклу**.

Інструкції циклу з передумовою відповідає блок-схема на рис. 6.1.

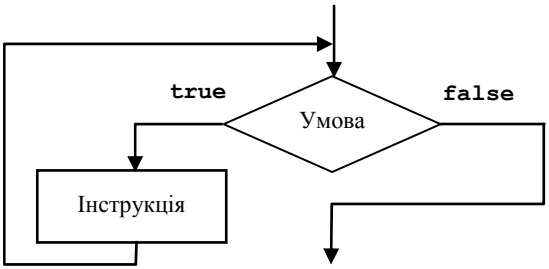


Рис. 6.1. Блок-схема інструкції циклу з передумовою

Умову в інструкції циклу називають **умовою продовження**, оскільки, якщо вона істинна, то виконання інструкції циклу продовжується. Цикл починається обчисленням умови, тому її ще називають **передумовою**.

Інструкції циклу з передумовою застосовують зазвичай тоді, коли кількість повторень циклу *наперед невідома*.

Приклади

1. За цілим $a \geq 0$ обчислити найменше n, за якого $n! > a$.

Будемо в циклі обчислювати послідовні значення $n!$, поки вони не стануть більше a. Коли цю умову буде порушено, n буде на 1 більше потрібного значення.

```
int minArgFact(int a)
{
  int n=1, fact=1;
  while (fact<=a) { fact*=n; n=n+1;}
  // fact=(n-1)!, fact > a
  return n;
}
```

Зауважимо, що за $a \geq 12!$ (це число порядку 500 мільйонів) замість 13! обчислюється помилкове значення, оскільки 13! не є числом типу `int`.

2. Обчислити суму цифр заданого натурального числа.

Щоб знайти суму цифр заданого числа n, можна діяти так. Спочатку сума дорівнює 0. Далі беремо молодшу цифру числа, додаємо її до суми й викреслюємо з числа. Повторюємо ці дії, поки в числі є цифри. Молодша цифра числа n є значенням виразу `n%10`, її викреслення можна зобразити як `n/=10`, наприклад `123%10 = 3`, `123/10 = 12`. Оформимо розв'язання функцією з параметром `n`, яка повертає обчислену суму цифр.

```
int digitsSum(int n)
{ int sum = 0; // початкова сума цифр
  while (n>0)
    { sum+=n%10; n/=10; }
  // n=0, у sum накопичено суму цифр
  return sum;
}
```

6.1.3. Збільшення та зменшення

У циклічних обчисленнях дуже часто використовуються присвоювання вигляду `x=x+1` та `x=x-1`. Їх можна задати в скороченій формі за допомогою одномісних операторів **збільшення** (інкременту) `++` і **зменшення** (декременту) `--`. Ці оператори (і

відповідні операції) мають **префіксну** (`++x`, `--x`) і **постфіксну** (`x++`, `x--`) **форми**.

Вираз із постфіксним оператором `x++` або `x--` змінює значення змінної `x` на `1`, але значенням самого виразу є значення `x` *перед зміною*. Вираз із префіксним оператором `++x` або `--x` теж змінює `x` на `1`, але значенням виразу є значення `x`, отримане *після зміни*. Ці відмінності виявляються, коли оператори `++` та `--` застосовуються всередині виразів.

Приклади

1. Інструкція `y=x++;` рівносильна `y=x; x=x+1;`, а інструкція `y=++x;` – `x=x+1; y=x;`. Якщо змінна `x` мала значення `1`, то після `y=x++` значенням `y` буде `1`, а після `y=++x - 2`. В обох ситуаціях значенням `x` стане `2`.

2. Цикл `while (k<=n){fact*=k; k=k+1;}` за допомогою оператора `++` можна записати в будь-якій з таких форм:

```
while (k<=n){ fact*=k; k++;}
while (k<=n){ fact*=k; ++k;}
```

◀

Операції `++` та `--` виконуються швидше ніж відповідні присвоювання вигляду `x=x+1` та `x=x-1`, тому рекомендується використовувати саме їх. Операції `++` та `--` застосовні до змінних будь-якого з базових типів, хоча найчастіше їх використовують із цілими змінними.

Скрізь, де немає необхідності використовувати старе значення змінної, рекомендується з виразів вигляду `n++` та `++n` вибирати `++n`, оскільки він виконується швидше й простіше.

Спосіб і порядок обчислення виразу залежить від компілятора, тому краще записувати операції збільшення або зменшення в окремих виразах або інструкціях, а не у складі інших виразів. Наприклад, значення виразів `(n++)*(n++)` та `(++n)*(++n)` у різних системах програмування навіть можуть відрізнятися. Гарантовано лише те, що до значення змінної `n` двічі додається 1.

Вправа 6.1. Що буде надруковано за програмою? Пояснити зв'язок між значеннями змінних `i` та `x` і `y`:

а)

```
#include <iostream>
using namespace std;
int main(){
 int i=1, x=1, y=2;
 while (x<y){
  i++; x*=i; y*=2;
  cout<<i<<" "<<x<<
        " "<<y<<endl;
 }
 system("pause");
 return 0;
}
```

б)

```
#include <iostream>
using namespace std;
int main(){
 int i=1, x=1, y=2;
 while (i<=10){
  i++; x*=i; y*=2;
  cout<<i<<" "<< x <<
        " "<<y<<endl;
 }
 system("pause");
 return 0;
}
```

6.1.4. Інструкція циклу з післяумовою

Інструкція циклу з **післяумовою**, або `do`-інструкція, має загальний вигляд

```
do інструкція while (умова);
```

Слово `do` (виконувати) є ключовим. Інструкція циклу з післяумовою виконується так. Спочатку виконується тіло циклу, потім обчислюється умова. Якщо вона хибна, то цикл завершується, інакше повторюється тіло й знову обчислюється умова. На відміну від інструкції циклу з передумовою, цикл *починається діями в тілі* й закінчується обчисленням умови.

Циклу з післяумовою відповідає блок-схема на рис. 6.2.

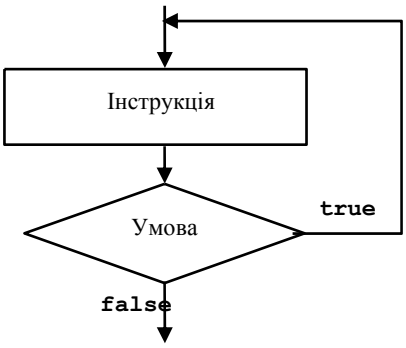


Рис. 6.2. Блок-схема інструкції циклу з післяумовою

Умова перевіряється після виконання тіла циклу, тому її називають **післяумовою**. Тіло циклу, заданого `do`-інструкцією, виконується обов'язково *хоча б один раз* (на відміну від `while`-інструкції).

Інструкцію циклу з післяумовою використовують, коли потрібно спочатку виконати тіло циклу, і лише потім перевіряти умову продовження.

Приклади

1. Потрібно з клавіатури ввести ціле число від 10 до 99. Якщо користувач набрав число за межами цього діапазону, то слід *повторити спробу*. Отже, спочатку треба вводити число, а потім перевіряти умову того, що число є двозначним.

```
do {
    cout << "Enter one integer in [10,99]>";
    cin >> k;
} while (!(10<=k && k<=99));
// 10<=k && k<=99
```

2. За двома натуральними числами n і m визначити, чи можна n подати як суму двох натуральних доданків, а m – як суму їх квадратів. Наприклад, $10 = 7+3$, $58 = 7^2 + 3^2$.

Доданків два, але для їх визначення достатньо одного циклу за можливими значеннями першого доданка x, а другий доданок визначається за умовою як $y = n-x$. Значення x перебираються, поки не знайдено розв'язок та x не більше y. Якщо після виходу з циклу x не більше y, то розв'язок знайдено, інакше розв'язків немає. Оформимо обчислення у вигляді функції, що повертає ознаку успішності пошуку й надає значення двом параметрам-посиланням (нулі, якщо пошук неуспішний).

```
bool twoItems(int n, int m, int & x, int & y){
  x = 0;
  do {
    x++; y = n-x;
  } while (x<=y && x*x+y*y!=m);
  if (x<=y) return true;
  else { x=0; y=0; return false; }
}
```

Інструкції циклу з післяумовою є в багатьох мовах програмування, але в деяких післяумова розуміється як умова завершення

(а не продовження) циклу. Тому при перекладі таких інструкцій іншими мовами *можуть виникати непорозуміння*.

Кожен цикл із післяумовою *можна замінити циклом з передумовою*, який в усіх мовах програмування розуміється однаково.

Наприклад, код із прикладу 1 (с. 108) набуде вигляду

```
k=1;    // щоб ініціювати введення, присвоїмо
        // змінній k "неправильне" значення
while (!(10<=k && k<=99)){
    cout << "Enter one integer in [10,99]>";
    cin >> k;
}
// у кінці циклу так само 10<=k && k<=99
```

6.1.5. Переривання та продовження циклу

Уперше інструкцію **break** було наведено в підрозд. 4.5, де за її допомогою закінчувалося виконання інструкції-перемикача. Виконання цієї інструкції всередині циклу будь-якого різновиду перериває й завершує цикл; далі виконуються дії, наступні за цим циклом.

Якщо **break** записано в інструкції циклу, вкладеній в іншу інструкцію циклу, то виконання **break** завершує вкладений цикл, а зовнішній цикл продовжується. Інструкція **continue** всередині циклу задає перехід на кінець тіла циклу. В інструкціях циклу з перед- і післяумовою після **continue** обчислюється умова продовження циклу.

Приклад 6.1. За допомогою клавіатури вводиться послідовність дійсних чисел. Потрібно підрахувати суму її додатних елементів, а за появи 0 видати накопичену суму й завершити роботу.

Запрограмуємо цикл, в якому вводиться й обробляється послідовність чисел. Уведене число зберігаємо в змінній **x**, а суму додатних елементів – у змінній **sum**. Якщо під час уведення трапилася помилка, то подальші дії з уведення не виконуються, а змінна **x** зберігає своє останнє значення (див. п. 2.7.2, с. 42). Тому умовою продовження циклу буде саме відсутність помилок уведення. (Інакше можна отримати цикл, який ніколи не завер-

шиться!) Цю умову задає значення виразу введення **cin>>x**, перетворене до логічного типу.

```
#include <iostream>
using namespace std;
int main()
{ double x;
  double sum=0;
  cout<<"Enter reals:\n";
  while (cin>>x){
    if (x==0.) break;    //виходимо з циклу
    if (x<0.) continue;  //пропускаємо від'ємні
    sum+=x;
  }
  cout << "sum=" << sum << endl;
  system("pause"); return 0;
}
```

prog014.cpp

Зазначимо, що використання інструкції **continue** в цій програмі є дуже штучним. Ще одним недоліком є те, що в кінці не повідомляється, чи були помилки під час уведення. Проте обробка помилок у вхідних даних виходить за межі цієї книжки. ◂

Інструкції програми виконуються *в порядку їх запису в програмі*. Про таку програму кажуть, що вона **структурована**. Інструкції **break** і **continue** *порушують* цей порядок обчислень, заплутуючи текст програми. Тому, користуючися ними, програміст повинен ретельно відслідковувати точку програми, якою продовжуються обчислення. Інколи ці інструкції дійсно скорочують запис розгалужень у циклі, проте в більшості випадків ті ж самі дії *можна описати без них*. Отже, не зловживайте **break** і **continue**.

Вправи

6.2. Модифікуйте програму prog014.cpp, щоб позбутися **break** і **continue**.

6.3. Написати функцію, що за цілим числом визначає:

а) кількість цифр його десяткового запису;

б) чи зустрічається в його десятковому запису задана цифра;

в) кількість входжень заданої цифри в його десятковий запис;

г) старшу цифру його десяткового запису;

д) мінімальну (максимальну) цифру його десяткового запису.

6.4. Написати функцію, що за цілим a обчислює й повертає найбільше n, за якого $n! \le a$.

6.5. Написати функцію, що за цілими n та m обчислює й повертає $[\log_n m]$. (У передумові до функції зверніть увагу на значення функції у випадку некоректних вхідних даних.)

6.6. На вхід програми дається послідовність символів `(` та `)`. Послідовність вважається правильною, якщо вона містить однакові кількості символів `(` і `)` та в довільному її початковому відрізку символів `(` не менше ніж `)`. Ознакою завершення послідовності є введення будь-якого непорожнього символу, відмінного від `(` та `)`. Програма має визначити, чи є вхідна послідовність правильною.

6.7. Програма в прикладі у п. 4.5 за введеним із клавіатури дійсним числом, знаком операції (`+`, `-`, `*` або `/`) і ще одним числом друкувала результат застосування операції до цих чисел. Модифікувати її, щоб після кожного обчислення вона запитувала користувача, чи треба виконати ще одне обчислення, і за згоди користувача виконувала його. Відповідь користувача промоделювати символьною змінною: значення `Y`, `y` відповідають необхідності подальших обчислень.

6.8. **Табулювання функції**. Нехай задано дійсні числа a, b, причому $a \le b$ ($[a; b]$ – відрізок, на якому виконується табулювання), і додатне дійсне число h – крок табулювання. Необхідно вивести на екран два стовпчики – перший задає послідовні значення a, $a+h$, $a+2h$, $a+3h$, …, b аргументу функції з відрізку $[a; b]$, а другий – значення функції $\sin x$ у відповідних точках. Незалежно від того, чи ділиться довжина відрізка націло на h, останній рядок повинен містити числа b та $\sin b$. Розв'язком має бути `void`-функція з параметрами `a`, `b`, `h`. У коментарі вказати передумови виклику функції та описати її поведінку за некоректних вхідних даних.

6.9. Написати функцію, що на відрізку $[a; b]$ із кроком h табулює обидві функції $\sin x$ і $\cos x$.

6.10. **Табулювання функції сторінками**. Написати функцію, що на відрізку $[a; b]$ табулює з кроком h функцію $\sin x$, але після виведення кожних m рядків виводиться запит, чи продовжувати друкування. Робота завершується після відповіді "0".

6.2. Інструкція циклу for

Приклад. Нагадаємо: значення факторіала $n!$ невід'ємного цілого числа n визначається формулою $n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$ при $n > 0$, $0! = 1$. Щоб обчислити цей добуток, можна взяти початкове значення 1 і послідовно помножити його на кожне число від 1 до n.

Цим діям відповідає послідовність інструкцій, в якій інструкції й вирази відіграють чітко визначені ролі.

```
fact=1;         // початкове значення добутку
k=1;            // початок перебирання множників
while (k<=n) { // перевірка умови продовження
  fact*=k;      // основна дія
  ++k;          // перехідна дія перед наступним кроком
}          // після закінчення повторень k=n+1, fact=n!
```

Ці елементи виконуються в тому самому порядку, якщо записати їх так:

```
fact=1;
for (k=1; k<=n; ++k) fact*=k;
```

Інструкція циклу **for**, або **for**-інструкція, має загальний вигляд

```
for (початкова дія; умова; перехідна дія)
   основна дія
```

Слово **for** зарезервоване, дужки та два знаки **;** усередині дужок є обов'язковими. Початкова дія, умова й перехідна дія є *виразами* (кожен із них може бути порожнім), основна дія – *інструкцією*. Тілом циклу **for** називають його основну дію. Інструкція **for** виконується так само, як і інструкції вигляду

```
початкова дія;
while (умова)
{
   основна дія;
   перехідна дія;
}
```

Інструкція **break** у тілі циклу **for** завершує його виконання, а інструкція **continue** завершує виконання лише тіла циклу; відразу після неї виконується перехідна дія.

Приклади

1. Дуже часто інструкція циклу **for** зустрічається у вигляді

```
for (k=0; k<n; ++k) інструкція
```

й задає виконання *інструкції* за значень $k = 0, 1, 2, \ldots, n\text{-}1$ або у вигляді

```
for (k=1; k<=n; ++k) інструкція
```

й задає виконання *інструкції* за значень $k = 1, 2, \ldots, n$, або у вигляді

```
for (k=n; k>0; --k) інструкція
```

й задає виконання *інструкції* за значень $k = n, n\text{-}1, \ldots, 2, 1$. Змінну `k` у цих ситуаціях інколи називають **лічильником циклу**.

2. Функція `printFraction` друкує перші k дробових знаків десяткового запису дійсного числа v від 0 до 1 (див. алгоритм у додатку А).

```
// pre: 0<=v<1 && k>0, інакше функція
// нічого не виводить
void printFraction(double v, int k){
   int d; //поточна цифра
   if (!(0<=v)&&(v<1)&&(k>0)) return;
   for(;k>0;k--){
     v*=10; d=int(v); v-=d;
     cout<<d;
   }
}
```

◀

Операція послідовного обчислення. У прикладі з обчислення факторіала спочатку треба присвоїти значення не тільки лічильнику `k`, але й змінній `fact`. На відміну від багатьох мов програмування, C++ дозволяє поєднати ці дві дії в одному виразі – за допомогою операції послідовного обчислення.

Операція зі знаком "`,`" позначає послідовне обчислення виразів, записаних через кому. Ця послідовність виразів розглядається як один вираз; його значенням є значення останнього виразу.

Запишемо з її використанням цикл обчислення факторіала:

```
for (fact=1,k=1; k<=n; k++) fact*=k;
```

Операція послідовного обчислення дозволяє на місці одного виразу записати кілька.

Вправи

6.11. За $n \geq 0$ значення $n!!$ (подвійний факторіал) задається так: $0!! = 1$, $1!! = 1$, $n!! = n \cdot (n\text{-}2)!!$, якщо $n \geq 2$. Написати функцію, що за цілим числом повертає його подвійний факторіал.

6.12. Написати функцію, що за заданими дійсними a, h і цілим m друкує значення функції $\operatorname{arctg}(\sin x)$ у точках a, $a+h$, …, $a+mh$.

6.3. Рекурентні співвідношення в циклічних алгоритмах

6.3.1. Рекурентні співвідношення

Повернемося до задачі обчислення факторіала невід'ємного цілого числа n. Для розв'язання цієї задачі серед інших варіантів було побудовано код

```
fact=1;
for (k=1; k<=n; k++) fact*=k;
```

Його можна прочитати й так:

```
факторіал числа 0 дорівнює 1;
поки не знайдемо факторіал числа n
  обчислювати факторіал кожного наступного числа
  як факторіал попереднього, множеного на це число;
//останнє знайдене значення факторіала є шуканим
```

Суттєвим у цьому алгоритмі є те, що шукається елемент послідовності факторіалів 0!, 1!, 2!, … з певним номером. При цьому сама послідовність визначається двома законами: один задає перший елемент послідовності, інший пояснює, як обчислити елемент за його номером і попереднім елементом. Отже, є початкові умови й рекурентне співвідношення, які разом задають послідовність. Для послідовності факторіалів початкові умови – це $0! = 1$, а рекурентне співвідношення – $n! = (n - 1)! \cdot n$ за $n > 0$.

Розглянемо загальнішу ситуацію. Припустимо, що перші k елементів послідовності $a_0, a_1, \ldots, a_n, \ldots$ задано явно (це **початкові умови** $a_0, a_1, \ldots, a_{k-1}$) і є закон F, який дозволяє за номером елемента та/або деякими попередніми елементами знайти всі інші елементи:

$a_i = F(i, a_{i-1}, \ldots, a_0)$ за всіх $i \geq k$.

Останню рівність називають **рекурентним співвідношенням**. Кожне рекурентне співвідношення разом із початковими умовами визначає певну послідовність.

Приклади

1. Рекурентне співвідношення $a_n = a_{n-1} + d$, $n \geq 1$, з початковими умовами $a_0 = a$ задає арифметичну прогресію з початковим елементом a та кроком d. Співвідношення $a_n = q \cdot a_{n-1}$, $n \geq 1$, та

умови $a_0 = a$ визначають геометричну прогресію з початковим елементом a та коефіцієнтом q.

2. Рекурентне співвідношення $S_n = S_{n-1} + \sin(n)$, $n \geq 1$, з початковими умовами $S_0 = 0$ задає суму $\sum_{k=1}^{n} \sin(k)$.

3. Рекурентне співвідношення $F_n = F_{n-1} + F_{n-2}$, $n \geq 2$, з початковими умовами $F_0 = 0$, $F_1 = 1$ визначає рекурентну послідовність, що має назву **послідовність Фібоначчі**. Вона має такий початок: 0, 1, 1, 2, 3, 5, 8, 13, … . ◂

6.3.2. Програмування циклічних обчислень за рекурентними співвідношеннями

За допомогою рекурентних співвідношень можна розв'язати чимало задач. Головне – знайти рекурентне співвідношення й початкові умови. Далі безпосередньо за ними неважко запрограмувати циклічні обчислення.

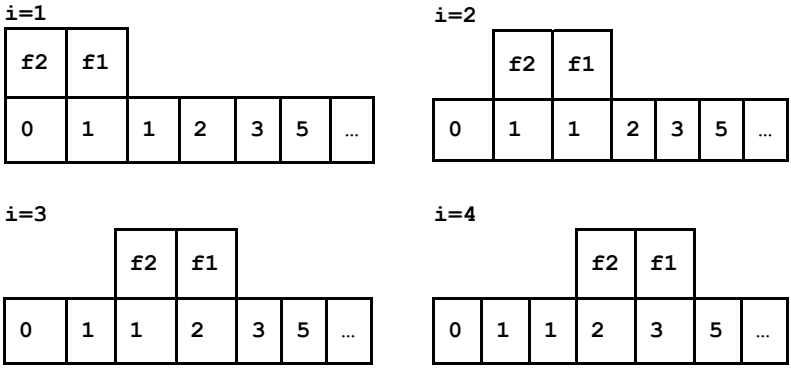
Приклад. За цілим числом $n \geq 0$ знайти число Фібоначчі F_n.

Для розв'язання цієї задачі запрограмуємо обчислення за рекурентним співвідношенням $F_n = F_{n-1} + F_{n-2}$, $n \geq 2$, і початковими умовами $F_0 = 0$, $F_1 = 1$.

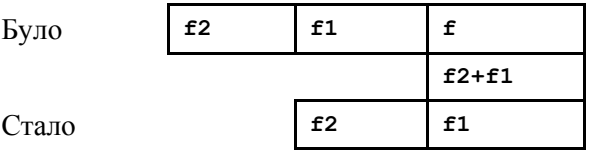
Якщо $n = 0$, то результатом є 0; якщо $n = 1$, то результатом є 1. За $n \geq 2$ будемо обчислювати елементи послідовності один за одним, доки не отримаємо числа із заданим номером. За F_0 та F_1 обчислимо F_2, за F_1 та F_2 – F_3 і т. д. При цьому, щоб обчислити наступний елемент, потрібно знати два останні елементи, що йому передують. Щоб вчасно зупинитися, потрібно також зберігати й щоразу збільшувати номер елемента, обчисленого останнім.

Зверніть увагу: щоб знайти n-й елемент послідовності, заданої рекурентним співвідношенням, необхідно послідовно обчислювати 1-й, 2-й, ..., n-й елементи.

Отже, виділимо поняття: *передостаннє число*, *останнє* й *наступне*, а також *номер* числа, обчисленого останнім. Зобразимо їх змінними `f2`, `f1`, `f` та `i`, відповідно. Змінні `f2`, `f1` можна уявити як віконця, що пересуваються послідовністю Фібоначчі.



Отже, зі збільшенням `i` віконце пересувається на одну позицію праворуч. Для цього обчислюється наступний елемент і зберігається в змінній `f`, значення `f1` стає значенням `f2`, а `f` – значенням `f1`.



Реалізуємо наведені міркування у функції.

```
int fibo(int n) {
//pre: n>=0, інакше повертаємо -1
  int f2=0, f1=1, f;
  if (n<0)return -1;
  if (n==0) return f2;
  if (n==1) return f1;
  int i;
  for (i=1; i<n;i++) {
    f=f1+f2;
    f2=f1; f1=f;     //зсув віконця
  }
  return f;
}
```

Увага! Під час зсуву віконця першою нове значення має отримувати його "найстаріша" змінна, тобто спочатку `f2`, а потім `f1`. Присвоювання `f1=f; f2=f1;` були б дуже грубою помилкою.

Для перевірки цієї функції потрібно, наприклад, у головній функції програми задати аргументи в її виклику -1, 0, 1, 2 та деякі інші так, щоб було перевірено обробку некоректних параметрів, обчислення початкових елементів, одно- й багаторазове виконання тіла циклу.

Зауважимо: послідовність чисел Фібоначчі зростає доволі швидко, наприклад, 50-те число не вміщається в типі **`int`**. ◂

У наведеному прикладі в інструкції циклу із заголовком **`for(i=1; i<n; i++)`** змінна **`i`** використовується тільки в тілі циклу. Мова C++ дозволяє такі змінні *оголошувати в початковій дії циклу* **`for`**. Це оголошення буде діяти до кінця тіла циклу. Відповідний фрагмент коду такий:

```
for (int i=1; i<n;i++) {
  f=f1+f2;
  f2=f1;f1=f;//зсув віконця
} //оголошення імені i вже не діє
```

Рекурентні співвідношення використовують у програмуванні обчислень, пов'язаних із послідовностями. Співвідношення дозволяють формально виразити залежності між величинами, які обчислюються послідовно, а на основі цих виразів легко запрограмувати цикл. Для розв'язання задачі треба:

а) зрозуміти, що величини утворюють послідовність;

б) записати відповідне рекурентне співвідношення;

в) визначити перші члени послідовності, що обчислюються без співвідношення (за початковими умовами);

г) сформулювати умову продовження, за якої застосовується співвідношення.

Після цього згадані вище дії й порядок їх розташування в алгоритмі стають очевидними.

Вправи

6.13. Яким буде значення виразу **`fibo(5)`**, якщо зсув віконця записати в іншій послідовності, тобто як **`f1=f; f2=f1;`**?

6.14. Написати функцію, що за цілим $n \geq 0$ повертає n-й елемент послідовності, заданої рекурентним співвідношенням $a_n = 3a_{n-1} - a_{n-2}$, $n \geq 3$, $a_1 = 3$, $a_2 = -2$.