## ОБРОБКА ТА ОТРИМАННЯ ПОСЛІДОВНОСТЕЙ

### 8.1. Послідовності псевдовипадкових чисел

У багатьох задачах (від моделювання природних або соціальних процесів до розкладання карт) потрібні послідовності чисел, що належать певній множині, але більше ніяк не пов'язані одне з одним. Такі числа називаються випадковими. Послідовності випадкових чисел часто імітують, використовуючи **генератор псевдовипадкових чисел** – підпрограму, яка за певним алгоритмом утворює послідовність чисел, що виглядає випадковою. Щоб кожного разу отримувати різні послідовності, алгоритм ініціалізують числом, яке зазвичай визначає всю послідовність.

Мовою С++ послідовності псевдовипадкових чисел можна отримувати за допомогою двох функцій, оголошених у файлі **cstdlib**. Функція з прототипом **void srand(unsigned int);** установлює початкове значення послідовності, використовуючи для цього аргумент у виклику (ним має бути невід'ємне ціле число). Функція з прототипом **int rand(void);** обчислює й повертає наступне псевдовипадкове значення. Усі значення є цілими числами в діапазоні від 0 до 32767. Число 32767 позначається константою **RAND\_MAX**, яку теж оголошено у файлі **сstdlib**.

Ім'я **rand** є скороченням слова *random* (випадковий), а **srand** – скороченням *seed random* (засіяти випадкове). Значення аргументу у виклику **srand** називається **зерном**, оскільки "зерно" є головним значенням англійського слова *seed*.

Розглянемо приклад програми, що отримує від користувача зерно, створює тисячу псевдовипадкових дійсних чисел у діапазоні [0; 1] та обчислює їх середнє арифметичне (воно має бути досить близьким до 0,5). Дійсні числа з [0; 1] утворюються шляхом ділення цілих псевдовипадкових чисел на **RAND\_MAX**.

**#include <iostream>**

**#include <cstdlib> using namespace std; int main() { double sum; int k, seed;**

**cout << "Enter int seed>"; cin >> seed; srand(seed);**

**for(k=0,sum=0; k<1000; ++k) sum += rand()/double(RAND\_MAX); cout << sum/1000 << endl; system("pause"); return 0;**

**}**

**prog019.cpp**

Зерно можна зробити випадковим за допомогою функції **time**, яку оголошено у файлі **time.h**. Для цього потрібен виклик **srand(unsigned(time(NULL)));**

Функція **time** звертається до вбудованого годинника й отримує від нього число, пов'язане з поточним днем і часом.

#### Вправи

8.1. Напишіть функцію, що за кожного свого виклику повертає наступне псевдовипадкове дійсне число з відрізка [*a*; *b*].

8.2. Напишіть функцію, що за кожного свого виклику повертає наступне псевдовипадкове ціле число з проміжку [*a*; *b*]. *Вказівка*. Перетворити число *x*, наприклад, з проміжку від 0 до

100 на число від -5 до 4 можна за формулою *x*%(4-(-5)+1)+(-5).

### 8.2. Статична пам'ять

Нагадаємо (див. підрозд. 4.4): ім'я, оголошене зовні функцій, називається **глобальним**. Змінні, оголошені зовні функцій, існують протягом усього часу виконання програми та зберігаються в окремій області пам'яті. Область пам'яті зі змінними, що існують протягом усього часу виконання програми, і самі ці змінні називаються **статичними**. Статичними можуть бути як глобальні змінні, так і локальні.

**Приклад**. Розглянемо програму, в якій статичною є саме локальна змінна.

**#include <iostream> using namespace std; int nCalls() { static int i=0; return ++i;**

**} int main(){**

**cout << nCalls() << endl; cout << nCalls() << endl; system("pause"); return 0;**

**}**

**prog020.cpp**

Специфікатор **static** в означенні **static int i=0;** указує, що:

* пам'ять для змінної **i** виділяється в статичній пам'яті програми **на початку виконання програми** та змінна відразу ініціалізується значенням **0**;
* у викликах функції **nCalls** для змінної **i** **пам'ять** **не виділяється**;
* між викликами функції **nCalls** змінна **i** **зберігає своє значення**.

Кожен виклик функції **nCalls** збільшує значення змінної **i** на **1** і повертає його. Отже, за першого виклику функція поверне 1, за другого – 2 тощо. Узагалі, функція **nCalls** повертає кількість своїх викликів.

Специфікатор **static** не впливає на область дії оголошення імені. Ім'я **i** оголошено в тілі функції **nCalls**, тому ім'я змінної **i** є локальним у функції, а за межами функції **nCalls** ця змінна недоступна. 

Локальні статичні змінні варто використовувати, коли необхідно забезпечити недоторканність значень змінних функції між її викликами.

**Вправа 8.3**. Написати функцію, кожен виклик якої повертає наступне число Фібоначчі, тобто перший виклик – 0, другий – 1, наступні – 1, 2, 3, 5 тощо.

### 8.3. Обробка послідовностей

#### 8.3.1. Загальна схема побудови програми

У прикладі 6.1 на с. 109 розбиралася задача: визначити суму додатних чисел у послідовності, що вводиться з клавіатури й закінчується числом 0, яке до послідовності не входить. Для цієї задачі характерним є те, що:

* входом є послідовність чисел, кількість яких наперед невідома;
* вхід містить ознаку завершення послідовності (число 0).

Ця задача є типовою, тому побудуємо її розв'язання так, щоб його можна було пристосувати до інших аналогічних задач із мінімальними змінами. Сформулюємо алгоритм розв'язання в загальному вигляді.

**Ініціалізація**

**Головний робочий цикл, а саме: while (є наступний елемент) обробити наступний елемент Завершення**

**Ініціалізація** – це дії, підготовчі до отримання вхідної послідовності (наприклад, вивести запрошення для користувача) і до обробки введених елементів (наприклад, установити в 0.0 суму введених додатних чисел). **Головний робочий цикл** – це повторювані дії з отримання й обробки елементів послідовності. **Завершення** – це заключні обчислення та, можливо, дії, потрібні, щоб коректно закінчити обробку послідовності.

Розглянемо головний робочий цикл детальніше. Кожен фрагмент коду повинен мати один обов'язок, тому отримання елементів послідовності слід, наскільки це можливо, відокремити від їх обробки. Отримання наступного елемента "сховаємо" у функцію. Джерелом, з якого надходить послідовність, може бути не тільки клавіатура, але за межами функції *від джерела ніщо не залежатиме*.

Якою має бути ця функція? Вона має повертати наступний елемент послідовності, якщо він є, а якщо немає – то якось про це повідомляти. Вона також має повідомляти про неуспішність спроби отримати наступний елемент, тобто про помилку. Отже, функція має повертати **статус останньої операції** отримання елемента послідовності та сам цей елемент. Статус будемо повертати як значення функції, а елемент – за допомогою параметра-посилання. Домовимося: якщо отримати наступний елемент послідовності неможливо, то функція не змінює значення параметра. Отже, прототип функції буде таким: **int get (double &x);**

Розглянемо використання функції **get** у головному циклі. Значення, які може повернути функція **get**, позначимо іменами **OK** (елемент отримано), **EOS** (скорочення від *end of sequence* – послідовність вичерпано) та **ERROR** (спроба отримати елемент неуспішна). Для того, щоб обробити помилку в заключних обчисленнях, будемо зберігати статус виклику **get** у допоміжній змінній **state** (стан). Отже, основний робочий цикл буде таким:

**while ((state=get(x))==OK) обробити x;**

Продовжимо відокремлювати обчислення від повідомлень. Обчислення реалізуємо в окремій функції **process**, а виведення результатів – у головній функції. Функція **process** повертає ознаку успішності обробки послідовності (**OK**, якщо все гаразд, інакше **ERROR**) та обчислену суму за допомогою параметрапосилання. Головна функція викликає **process**, отримує від неї результати й виводить відповідні дані на екран.

Реалізуємо функцію **get** за умови, що послідовність дійсних чисел вводиться з клавіатури, а число 0 вважається ознакою кінця послідовності.

**#include <iostream> using namespace std; const int OK=0, EOS=1, ERROR=2;**

**int get(double &x) // отримати наступний елемент**

**{ double y;**

**cout<<"Enter one real (0 to finish):\n"; if (!(cin>>y)) return ERROR; if (y==0.0) return EOS; x=y; return OK;**

**}**

**int process(double &sum) // розв'язання задачі**

**{ double x;**

**int state=OK; sum=0; //ініціалізація while((state=get(x))==OK){ //є поточний елемент if(x>0.) sum+=x; //обробити поточний елемент**

**}**

**// state == EOS || state == ERROR if (state == ERROR) return ERROR; else return OK;**

**} int main()**

**{ double sum; int state; state = process(sum); // виведення результатів if (state==ERROR) cout<<"An ERROR exists.\n"; else**

**cout << "sum=" << sum << endl; system("pause"); return 0;**

**}**

**prog021.cpp**

Наведена структура програми дозволяє за необхідності легко змінити джерело послідовності, умову її завершення або тип вхідних даних, оскільки для цього доведеться модифікувати тільки функцію **get**. Зміна ж дій, виконуваних з елементами послідовності, потребує змін тільки у функції **process**.

#### 8.3.2. Зміна умови завершення послідовності

Розглянемо попередню задачу за умови: ознакою закінчення послідовності є число, що збігається з першим і до послідовності не входить. Наприклад, послідовність 1, 3, 4 задається входом **1 3 4 1**. Щоб визначити, чи досягнуто кінець послідовності, у функції **get** будемо зберігати перше введене значення (у змінній **first**). Оскільки перше значення обробляється не так, як наступні, то необхідно знати, вводиться перший чи не перший елемент послідовності. Ознаку першого елемента виразимо логічною змінною **isFirst**. Змінні **first** та **isFirst** мають бути доступними лише у функції **get** і зберігати своє значення між її викликами, тому зробимо їх **статичними**. Отже, змінимо лише функцію **get**.

**int get(double &x){ static bool isFirst=true; static double first; double y;**

**if (isFirst) cout<<"Enter real:\n"; else cout<<"Enter real ("<< first<<" to finish):\n"; if (!(cin>>y)) return ERROR; if (isFirst)**

**{ isFirst=false; first=y; x=y; return OK; } else**

**{ if (y==first) return EOS; x=y; return OK;**

**}**

**}**

#### 8.3.3. Зміна дій, виконуваних із послідовністю

Розглянемо, як змінюється програма, якщо з послідовністю виконуються інші дії.

**Приклад: максимальний елемент**. Необхідно знайти максимальне число послідовності. Отримуємо перше число – спочатку воно максимальне, тому збережемо його в змінній **max**. Далі в головному робочому циклі отримуємо наступні елементи, порівнюємо їх зі значенням **max** і за необхідності змінюємо його.

У цій задачі є відмінність від попередньої: у порожній послідовності сума чисел нульова, але максимального елемента немає. Тому, якщо послідовність порожня, функція **process** має повернути ознаку цього факту – значення **EMPTY**. Отже, результатом обробки послідовності є одна з ознак: **ERROR** – виникла помилка, **EMPTY** – послідовність порожня, **OK** – знайдено максимальний елемент. Додамо до інших імен констант ім'я **EMPTY**. **const int OK=0, EOS=1, ERROR=2, EMPTY=1;**

Згідно з наведеними міркуваннями змінимо функцію **process** і головну функцію.

**int process(double &max)**

**{ double x; int state=OK; state=(get(max)); if (state==EOS) return EMPTY; if (state==ERROR) return ERROR;**

**// помилки немає, послідовність непорожня while ((state=get(x))==OK){ if (x>max) max=x;**

**} // state==EOS || state==ERROR if (state==ERROR) return ERROR; return OK;**

**} int main()**

**{ double max; int state; state=process(max); if (state==ERROR) cout<<"An ERROR exists.\n"; else if (state==EMPTY) cout<<"The sequence is EMPTY.\n"; else**

**// помилки немає, послідовність непорожня cout << "max=" << max << endl; system("pause"); return 0;**

**}**

**Приклад: монотонна послідовність**. Необхідно визначити, чи є вхідна послідовність монотонною. Розв'яжемо загальнішу задачу: визначимо тип непорожньої послідовності – стаціонарна або нестаціонарна, а саме: неспадна, незростаюча або немонотонна. Ознакам цих типів дамо імена відповідно **CONSTANT, UP, DOWN** та **OTHER**. Одне з цих значень функція **process** має повернути за допомогою свого параметра-посилання **sort** (тип), якщо отримано хоча б один елемент послідовності.

Алгоритм обробки послідовності такий. Отримаємо перший елемент послідовності. Послідовність з одного елемента є стаціонарною, тому змінна **sort** має значення **CONSTANT**. Далі, нехай оброблений початок послідовності має тип **sort**, відомий попередній елемент послідовності **prev** і отримано новий елемент **x**. Залежно від співвідношення між значеннями **prev** та **x** тип послідовності змінюється, як це вказано в таблиці. В інших ситуаціях тип послідовності не змінюється, тому ці ситуації не вказано.

|  |  |  |  |
| --- | --- | --- | --- |
| Значення **sort** | Співвідношення | Нове **sort** | значення |
| **CONSTANT** | **prev>x** | **DOWN** |  |
| **prev<x** | **UP** |  |
| **UP** | **prev>x** | **OTHER** |  |
| **DOWN** | **prev<x** | **OTHER** |  |

Запишемо означення типів послідовності й функцію **process. const int CONSTANT=0, UP=1, DOWN=-1, OTHER=2; int process(int &sort)**

**{ double prev, x; int state=OK; state=get(prev); if (state==EOS) return EMPTY; if (state==ERROR) return ERROR; sort=CONSTANT; while ((state=get(x))==OK){ switch (sort) { case CONSTANT: if (prev>x) sort=DOWN; else if (prev<x) sort=UP; break;**

**case UP: if (prev>x) sort=OTHER; break; case DOWN: if (prev<x) sort=OTHER; break;**

**} prev=x;**

**} // state==EOS || state==ERROR if (state==ERROR) return ERROR; return OK;**

**}**

Перевагою цього коду є те, що він *цілком прозоро задає логіку перевірки послідовності*. Автори бачили багато помилкових програм перевірки послідовності на монотонність. Найгіршим у них було те, що спроби виправити помилки спричиняли появу інших помилок, код заплутувався, і вихід був лише один – почати все з нуля.

"Родзинкою" наведеного способу перевірки послідовності є те, що значення змінної **sort** фактично *визначає стан процесу обробки послідовності й керує цим процесом*. Саме за станом вирішується, яке зі співвідношень між попереднім і наступним елементом слід перевіряти, а результат цієї перевірки визначає новий стан обробки.

Підхід із використанням станів застосовний до великої кількості задач програмування.