Лекція 11. Потоки введення-виведення даних. Використання файлів

1. *Потоки даних.*
2. *Виведення в cout.*
3. *Введення з cin.*
4. *Введення-виведення файлів*

Засоби введення-виведення (засоби I/O) не є частиною мови С. Їхню реалізацію віднесено до відповідальності розробника компілятора. (Аби надати йому повну свободу вибору того способу, який найкраще враховує особливості конкретної платформи.) У мові С++ з цією метою використано концепцію потоків, для взаємодії з якими створено бібліотеку класів. На сьогодні ця бібліотека стандартизована, тобто, де-факто введена у стандарт мови.

З точки зору програми введення та виведення – це потік байтів, що надходять з пристрою введення до програми і з програми до пристрою виведення. Потік пов'язує джерело та приймач байтів. Стандартний пристрій введення – клавіатура, інший часто вживаний пристрій – диск. Стандартний пристрій виведення – екран, інші – друкарка, файл, канал зв'язку. Програма забирає байти з потоку введення і вставляє байти в потік виведення. Для пришвидшення роботи з пристроями потоки використовують буфери обміну – спеціальні ділянки пам'яті, що тимчасово накопичують блоки даних для обміну з пристроєм. Обмін даними з потоком може потребувати перетворення формату даних. Наприклад, введені з клавіатури символи-цифри треба перетворити на число, і, навпаки, щоб вивести число на екран треба перетворити число до символьного вигляду.

Взаємодію з потоком, керування буфером, перетворення даних виконують об'єкти класів *iostream* та *fstream*. Директива *#include <iostream>* робить доступними в програмі вісім потокових об'єктів для роботи з потоками восьми- та шістнадцятирозрядних символів (імена об'єктів для розширеного набору символів типу *w_char* починаються літерою *w*):

- стандартний потік введення *cin*, *wcin*;
- стандартний потік виведення *cout*, *wcout*;
- небуферизований потік виведення помилок *cerr*, *wcerr*;
- потік виведення помилок з накопиченням у буфері *clog*, *wclog*.

Виведення в *cout* виконує оператор вставки << з прототипом *ostream& operator<<(type)*. Він перевантажений для всіх вбудованих типів. Програміст може перевантажити його для власних класів (тим самим задати перетворення об'єкта до текстового вигляду). Зауважимо, що *cout* є екземпляром класу *ostream*.

Є ще два додаткові методи виведення: окремого символа *ostream& put(char)* і виведення рядка *ostream& write(strAddr, strSize)*. Останній просто пересилає в потік *strSize* байтів, починаючи з розташованого за адресою *strAddr*.

Якщо потрібно очистити буфер потоку виведення і гарантовано доправити дані до пристрою виведення, використовують маніпулятор *flush*, наприклад, *flush(cout)*.

Форматування виведення в *cout*. Для побудови текстового вигляду значень вбудованих типів об'єкт використовує певний формат за замовчуванням: ширина поля виведення дорівнює кількості символів текстового зображення, незначащі нулі не друкуються, крапка в записі дійсного числа друкується тільки тоді, коли дробова частина відмінна від нуля.

Формат за замовчуванням часто не відповідає нашим потребам, зокрема, тому, що числа займають поля різної ширини, постійно доводиться виводити розділювачі між значеннями. Розглянемо засоби керування форматом виведення.

- Встановлення ширини поля виведення. *cout.width()* повертає поточну встановлену ширину; *cout.width(anInt)* встановлює ширину *anInt* символів для одного наступного виведення – після виведення ширина автоматично повертається до встановленої за замовчуванням, тому ширину задають для кожного виведення окремо. Замала задана ширина автоматично збільшується до достатнього розміру.

- Встановлення символа-заповнювача. За замовчуванням порожні позиції поля виведення заповнюють пропуски. Цю установку можна змінити: *cout.fill(aChar)*. Діє до наступної зміни.
- Задання кількості цифр після десяткової крапки: *cout.precision(numDigits)*. Діє до зміни.
- Для керування іншими режимами використовують метод *setf()* і маніпулятори.

Метод *setf()* встановлює прапорці об'єкта *cout* за допомогою констант-бітових масок, визначених в класі *ios_base*. Його прототип *fmtflags setf(fmtflags)*. Метод повертає поточний стан прапорців і встановлює новий (старий можна зберегти і використати пізніше для повернення до попереднього стану). Встановлення прапорця має вигляд *cout.setf(flagConst)*. Аргументами можуть бути такі константи:

ios_base::boolalpha – друкувати значення типу bool словом, а не числом;
ios_base::showbase – друкувати префікс, що відображає основу системи числення;
ios_base::showpoint – друкувати десяткову крапку для дійсних чисел;
ios_base::uppercase – друкувати великі букви для шістнадцяткових і дійсних;
ios_base::showpos – друкувати знак + перед додатними числами.

Відмінити зроблені установки можна методом *void unsetf(flagConst)*.

Є ще один перевантажений метод *setf()* з прототипом *fmtflags setf(fmtflags, fmtflags)*. Його використовують з такими парами констант:

перший аргумент		*другий аргумент*	
ios_base::dec *ios_base::oct* *ios_base::hex*	десяткова вісімкова шістнадцяткова	*ios_base::basefield*	встановити основу системи числення
ios_base::fixed *ios_base::scientific*	фіксована плаваюча	*ios_base::floatfield*	формат відображення десяткової крапки
ios_base::left *ios_base::right* *ios_base::internal*	до лівого краю до правого краю знак, префікс ліворуч, число – праворуч	*ios_base::adjustfield*	встановити спосіб вирівнювання

Таким чином, щоб надрукувати *double x* в полі шириною 10 знаків з чотирма знаками після коми, крапкою і знаком +, потрібно запрограмувати наступне:

```
double x=7.5;
cout.setf(ios_base::showpoint | ios_base::showpos);
cout.setf(ios_base::fixed, ios_base::floatfield);
cout.precision(4); cout.width(10); cout<<x<<'\n';
 // отримано на екрані:
    +7.5000
```

Використання маніпуляторів дещо полегшує керування об'єктом *cout*. Воно має дві форми: *manipul(cout)* або *cout<<manipul* (тут *manipul* – ім'я маніпулятора). Власне друга форма є зручною, бо дає змогу об'єднувати в один ланцюжок звертання до маніпуляторів і виведення даних. Перерахуємо імена маніпуляторів (за іменем легко зрозуміти призначення): *boolalpha, noboolalpha, showbase, noshowbase, showpoint, noshowpoint, showpos, noshowpos, uppercase, nouppercase, internal, left, right, dec, hex, oct, fixed, scientific*. Додаткові маніпулятори оголошено в заголовковому файлі *iomanip*: *setprecision(), setfill(), setw()* (від set width).

Введення даних за допомогою *cin* виконують оператором >>. Його перевантажено для всіх вбудованих типів (крім *bool*). Прототип методу: *istream& operator>>(type&)*. Метод пропускає розділювачі – пропуски, знаки табуляції, кінці рядка – аж до першого друкованого символа (навіть при введенні значень типу *char*), вибирає з потоку всі друковані символи (один символ для типу *char*) аж до наступного розділювача та перетворює отриманий рядок

на значення типу *type*, яке й присвоює аргументові. Програміст може перевантажити належним чином оператор >> для своїх класів. Зчитування перерветься також, якщо в потоці зустрінеться неправильний символ, наприклад, зірочка в записі числа. У цьому випадку змінюється стан потоку, і подальше введення припиняється аж до відновлення нормального стану.

Стан потоку визначається прапорцями *eofbit*, *badbit*, *failbit*. У нормальному стані вони рівні нулю. Досягнення кінця файла при введенні встановлює *eofbit*, введення неправильного символа – *failbit*, недіагностована помилка встановлює *badbit*.

Метод *clear()* скидає всі прапорці й повертає потік до нормального стану. Таку техніку ми вже використовували в прикладах у попередніх лекціях.

Методи для перевірки стану потоку: *bool good()* – поверне *true*, якщо всі прапорці рівні нулю, *bool eof()*, *boll bad()*, *bool fail()* – перевіряють відповідні прапорці. Перевірки дають змогу діагностувати причини зупинки введення і відкрити, якщо це можливо, потік для подальшого введення. Звичайно, при цьому потрібно якимось чином опрацювати причину помилки введення.

Нагадаємо, що для введення з *cin* можна використовувати й інші методи (як ми це вже робили раніше): *istream& get(char&)* та *char get()* читають один символ і не пропускають розділювачі; *istream& get(char*, int, char='\n')* та *istream& getline(char*, int, char='\n')* читають цілий рядок заданого розміру, або рядок до заданого термінального символа, залежно, що станеться швидше. Ці методи називають методами неформатованого читання, бо вони ніяк не перетворюють прочитані символи.

Метод *istream& ignore(int k=1, int t=EOF)* вилучає з потоку (без опрацювання) *k* символів або всі символи до термінального символа *t*.

Інші методи класу *istream*:

- *istream& read(addr, count)* читає *count* байт і записує їх без жодних перетворень за адресою *addr* – зручно для неформатованого читання;
- *char peek()* повідомляє черговий байт потоку введення, не забираючи сам байт з потоку – метод дає змогу «заглянути в майбутнє»: що там чергове в потоці, чи не термінальний символ;
- *int gcount()* повідомляє, скільки байтів прочитав останній виконаний метод неформатованого читання, наприклад, набагато ефективніше перевіряти довжину введеного рядка *cin.get(str,80)* так: *len=cin.gcount()*, ніж *len=strlen(str)*;
- *void putback(char)* поміщає символ у потік введення, наприклад, повертає в потік щойно прочитаний символ, або записує туди будь-який інший.

У багатьох випадках файлове введення-виведення для програм є набагато важливіше ніж консольне. З файла програма завантажує об'ємні вхідні дані, до файла записує отримані результати, у файлах зберігає проміжні, робочі дані. Обмін з файлами в C++ програмах виконують об'єкти класів *ofstream*, *ifstream*, породжених від *iostream*, тому всі методи, про які ми щойно говорили успадковуються і є доступними і для файлових операцій. Особливістю є те, що для роботи з файлами мало створити потоковий об'єкт, потрібно ще пов'язати його з конкретним фізичним файлом і вказати режими взаємодії з ним.

Запис даних до файла можна схематично зобразити таким фрагментом:

```
#include <fstream>       // підключити оголошення класів
ofstrem fout;            // створити потоковий об'єкт
fout.open("result.txt"); // асоціювати його з файлом (створити або стерти)
fout << value;           // виконати виведення (доступні засоби форматування)
fout.close();            // закрити потік, від'єднати його від файла
```

Оголосити об'єкт *fout* та приєднати його до файла можна і в одній інструкції за допомогою конструктора: *ofstream fout("result.txt");* В обох випадках потік відкривається з режимами за замовчуванням: файл створюється (якщо його ще не було) або витирається і відкривається як новий (якщо такий вже був на диску); файл текстовий – він міститиме всі дані у зовнішньому, символьному вигляді, оператор << виконуватиме перетворення даних.

У одній з попередніх лекцій ми оголосили функцію, яка вміє виводити на консоль таблицю значень заданої функції на заданому проміжку. Таблиця значень – це доволі цінний результат. Було б добре зберегти його до файла, щоб він не зникав разом з закінченням програми. Удосконалимо побудовану раніше функцію *Tabulate*: додамо до її параметрів ще один – посилання на потік виведення. Таке доповнення зробить її універсальною.

```
// Побудова таблиці значень функції
void TabulateToStream(std::ostream& os, func f, double a, double b, double h)
{
	unsigned n = round((b - a) / h);
	// виведення шапки таблиці
	os << "\t x\t|\t f(x)\n    ---------------------------------\n";
	// обчислення і виведення тіла таблиці
	for (unsigned i = 0; i <= n; ++i)
	{
		double x = a + i*h;
		(os << '\t').precision(4);
		(os << x << "\t|\t").precision(8);
		os << f(x) << '\n';
	}
	os << '\n';
}
```

Чи ви зауважили, які зміни відбулись? Серед параметрів з'явилося посилання на потік виведення. (Запам'ятайте: *потоки завжди передають через посилання!*) У блоці функції всі звертання до *cout* замінили на звертання до параметра *os* – довільного потоку виведення. Це всі зміни. А можливості *Tabulate* стали суттєво ширшими. Подивіться, як тепер її можна використати.

```
// файл для зберігання побудованих таблиць
ofstream fout("tables.txt");

// табулювання стандартної функції на консоль
cout << "\t\tTable 1. Sin(x)\n";
TabulateToStream(cout, sin, 0.0, M_PI_2, M_PI_2 / 12);
cin.get();
// а тепер - до файла
fout << "\t\tTable 1. Sin(x)\n";
TabulateToStream(fout, sin, 0.0, M_PI_2, M_PI_2 / 12);

// табулювання власної функції на консоль
cout << "\t\tTable 2. x^2+2x-3\n";
TabulateToStream(cout, [](double x) { return (x + 2.) * x - 3.; }, -4.0, 2.5, 0.25);
cin.get();
// а тепер - до файла
fout << "\t\tTable 2. x^2+2x-3\n";
TabulateToStream(fout, [](double x) { return (x + 2.) * x - 3.; }, -4.0, 2.5, 0.25);
// файлові потоки треба закривати
fout.close();
```

Виведення і в *cout*, і в *fout* продукує ідентичний текст. Ці об'єкти – «родичі», тому й мають однакові можливості. Відмінність є тільки в підготовці до використання: *cout* уже готовий, приєднаний до консолі, а файловий потік потрібно оголосити і приєднати до файла на диску.

Зчитування даних з файла схематично зобразимо так:

```
#include <fstream>       // підключити оголошення класів
ifstream fin;            // створити потоковий об'єкт
fin.open("data.txt");    // асоціювати його з файлом – він мав би існувати
fin >> var;              // виконати введення
fin.close();             // закрити потік, від'єднати його від файла
```

Як і раніше, оголошення та приєднання можна виконати разом: *ifstream fin("data.txt");* Режими за замовчуванням: файл існує; файл текстовий – він містить правильні зображення даних у зовнішньому, символьному вигляді, оператор >> виконуватиме перетворення даних до внутрішнього вигляду (двійкового коду).

У тій самій попередній лекції ми розв'язували таку задачу: задано 10-ти елементні масиви *a*, *b*, *c* дійсних чисел. Обчисліть величину $u = \begin{cases} (a,c), \text{ якщо } (a,a) > 5, \\ (b,c) \text{ у іншому випадку.} \end{cases}$ Пригадуєте, ми використовували функцію *ReadArray* для читання масиву з консолі. Це зручна функція, яка добре справляється зі своїм завданням, але введення з клавіатури значень елементів масиву великого розміру займатиме багато часу. Особливо, якщо програму потрібно виконати декілька разів. Щоб уникнути тривалих багаторазових уведень, запишемо всі числа до текстового файла і навчимо функцію читання масиву звертатися до файла.

```
// Поелементне введення масиву з потоку
void ReadArrayFromStream(num_type* a, size_t n, std::istream& is)
{
      for (size_t i = 0; i < n; ++i) is >> a[i];
}
```

Порівняно з *ReadArray* є трохи змін: з'явився параметр *is* – посилання на потік уведення (знову посилання!), звертання до *cin* замінили на звертання до *is*, з блока функції зникло виведення запрошення вводити числа. Справді, файл не вміє читати запрошення і не потребує їх, бо вже містить уведені числа. Числа до файла потрібно записувати так, як ми їх вводимо з клавіатури: через пропуск, з десятковою крапкою і знаком, якщо потрібно.

Нехай файл *data.txt* містить чотири рядки. У першому – кількість елементів масивів *a*, *b*, *c* згаданої задачі, у другому – значення елементів масиву *a*, у третьому і четвертому – масивів *b*, *c* відповідно. (Якщо масиви великі, то елементи можуть займати і декілька рядків аби тільки були записані в рядку через пропуск.) Уведення даних в програмі можна записати так:

```
ifstream fin("data.txt");
if (fin.is_open())
{
      // отримаємо з файла розмір масивів
      size_t n; fin >> n;
      num_type * a = new num_type[n];
      num_type * b = new num_type[n];
      num_type * c = new num_type[n];
      // завантажимо з файла масиви
      ReadArrayFromStream(a, n, fin);
      ReadArrayFromStream(b, n, fin);
      ReadArrayFromStream(c, n, fin);
      // введення завершено - файл треба закрити
      fin.close();
```

Для ясності продемонструємо можливий вміст файла *data.txt*:

```
5
4 5 3 1 2
2.5 3.5 1.3 -0.5 5.7
-1.1 0.2 1.5 4.3 1.1
```

Зауважимо, що функцію *ReadArrayFromStream* можна використовувати і для введення з консолі, якщо останнім аргументом при виклику вказати *cin*.

У ході зчитування даних з файла корисними будуть перевірки стану потоку методами *fin.good()*, *fin.eof()* тощо. Перед початком введення бажано перевіряти, чи справді відкрився вказаний файл (чи був він на диску) за допомогою *fin.is_open()*.

Закривання потоку методом *close()* є необов'язковим, бо потік закривається автоматично, коли потоковий об'єкт перестає існувати, але бажаним, бо це гарантує потрапляння даних до файла (як наслідок очищення буфера), економить ресурси системи, дає змогу повторно використовувати об'єкт для взаємодії з іншими файлами.

Перевантаження оператора >> і випробування зчитування об'єктів з файла залишимо читачеві як вправу.

Режими використання файла можна вказувати явно як другий аргумент методу відкривання файла. Можливими значеннями є такі бітові константи: *ios_base::in* (файл існує, відкриваємо для зчитування), *ios_base::out* (файл створюємо, відкриваємо для запису), *ios_base::ate* (встановити вказівник файла на кінець файла), *ios_base::app* (файл існує, відкриваємо для дописування), *ios_base::trunc* (усікти файл), *ios_base::binary* (файл двійковий).

Програма може зберігати дані у файлах не тільки в текстовому вигляді, але і в двійковому – у внутрішньому кодуванні. Звичайно, такі файли не вдасться прочитати, чи, тим більше, редагувати за допомогою редактора текстів, але двійкові файли мають ряд переваг. По-перше, введення-виведення відбувається швидше, бо не потребує виконання перетворень. По-друге, числові дані не зазнають спотворень через заокруглення. По-третє, дані у внутрішньому кодуванні майже завжди є набагато компактнішими.

Для роботи з двійковим файлом явно вказують режими використання, дані записують методом *write()*, зчитують – методом *read()*, як показано в наступних схематичних прикладах.

```
ofstream fout("planet.dat", ios_base::out | ios_base::binary);
struct planet {.....}; planet PL;
fout.write((char*) &PL, sizeof PL); // переслати до файла
                                    // "байтову копію" змінної

ifstream fin("planet.dat", ios_base::in | ios_base::binary);
planet Earth;
fin.read((char*) &Earth, sizeof Earth); // завантажити з файла
```

Наведемо ще один приклад використання двійкового файла: він може стати засобом обміну даними між програмами, написаними різними мовами програмування (оскільки компілятори використовують однакові способи кодування для даних однакових типів). Програма мовою C++ може бути ефективним обчислювачем, проте вона не надто зручна для відображення отриманих даних. Покладемо цей почесний обов'язок на інші програми. У нашому прикладі програма на C++ записує до файла послідовність дійсних чисел, програма на Object Pascal зчитує їх друкує як таблицю, програма на Python будує за ними графік. Задля контролю програма на C++ також відображає обчислені дані на екрані у вигляді таблиці.

mainFD.cpp

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <iomanip>

using std::cout; using std::ios_base;
using std::ofstream; using std::setw;

int main()
{
	// двійковий файл для зберігання результатів обчислень
	ofstream fout("table.dat", ios_base::out | ios_base::binary);
	double x, y;
	cout<<"    x    |    sin\n" <<"----------------------\n";

	for (int i = 0; i<=31; i++)
	{
		x = i * 0.2;
```

```
            y = sin(x);
            // задля контролю виводимо на консоль
            cout<<setw(6)<<x<<"   |   "<<y<<'\n';
            // задля збереження записуємо до файла
            fout.write((char*) &x, sizeof x);
            fout.write((char*) &y, sizeof y);
        }
        // готово, Файл обов'язково закриваємо
        fout.close();
        system("pause");
        return 0;
}
```

TableFromFile.dpr

```
program TableFromFile;

{$APPTYPE CONSOLE}
{$R *.res}

uses
  System.SysUtils;

var f:file of real;
    x, y: real;
    i: integer;
begin
  assignFile(f,'..\..\..\Files\table.dat'); reset(f);
  writeln('     x   |     sin');
  writeln('-----------------------');
  while not eof(f) do
  begin
    read(f, x, y);
    writeln(x:6:1, '   |', y:12:6)
  end;
  closeFile(f);
  readln
end.
```

PythonApplication.py

```
import struct
import matplotlib.pyplot as plt

file = open(r'..\Files\table.dat', 'rb')
xList = [] # список значень аргумента (абсциси)
yList = [] # список значень функції   (ординати)
while True:
    data = file.read(16)
    if data == b'': break # досягли кінця файла
    # двійкові дані треба перетворити на дійсні числа
    x, y = struct.unpack('2d', data)
    xList.append(x)
    yList.append(y)
# графічне відображення прочитаних чисел
plt.plot(xList, yList)
plt.title('Plot of y=sin(x) for x=0(0.2)6.2')
plt.show()
```

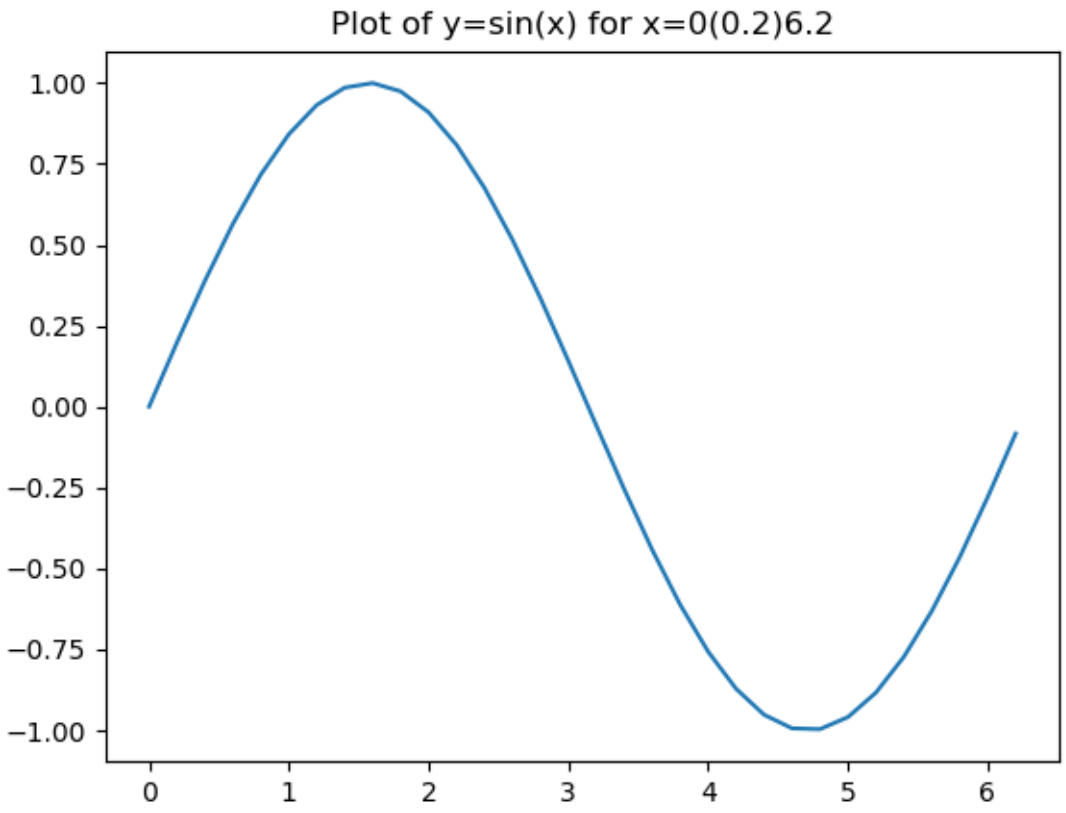
Ми не будемо тут обговорювати програмування мовою Pascal чи мовою Python. Просто порівняйте текстове виведення різних програм і можливості графічного відображення даних.

By C++ program

x	sin
0	0
0.2	0.198669
0.4	0.389418
0.6	0.564642
0.8	0.717356
1	0.841471
1.2	0.932039
1.4	0.98545
1.6	0.999574
1.8	0.973848
2	0.909297
2.2	0.808496
2.4	0.675463
2.6	0.515501
2.8	0.334988
3	0.14112
3.2	-0.0583741
3.4	-0.255541
3.6	-0.44252
3.8	-0.611858
4	-0.756802
4.2	-0.871576
4.4	-0.951602
4.6	-0.993691
4.8	-0.996165
5	-0.958924
5.2	-0.883455
5.4	-0.772764
5.6	-0.631267
5.8	-0.464602
6	-0.279415
6.2	-0.0830894

By Object Pascal program

x	sin
0.0	0.000000
0.2	0.198669
0.4	0.389418
0.6	0.564642
0.8	0.717356
1.0	0.841471
1.2	0.932039
1.4	0.985450
1.6	0.999574
1.8	0.973848
2.0	0.909297
2.2	0.808496
2.4	0.675463
2.6	0.515501
2.8	0.334988
3.0	0.141120
3.2	-0.058374
3.4	-0.255541
3.6	-0.442520
3.8	-0.611858
4.0	-0.756802
4.2	-0.871576
4.4	-0.951602
4.6	-0.993691
4.8	-0.996165
5.0	-0.958924
5.2	-0.883455
5.4	-0.772764
5.6	-0.631267
5.8	-0.464602
6.0	-0.279415
6.2	-0.083089



Додаткові можливості керування вказівником файла.

```
fstream f("MyFile.txt");

long pos = f.tellg(); // повертає номер позиції вказівника файла (в байтах),
                      // відрахованої від початка файла
long newPos = pos +- x;
f.seekg(newPos);      // встановлює вказівник файла в нову (абсолютну)
                      // позицію, відраховану від початка файла
// або
f.seekg(offset, ios::beg або ios::cur або ios::end);
        // зміщує вказівник файла на offset байтів відносно
        // початку файла, або поточної позиції, або кінця файла
```

Керування вказівником файла застосовують тоді, коли хочуть змінити звичайний послідовний спосіб читання даних з файла.

Література

1. Бублик В.В. Об'єктно-орієнтоване програмування мовою С++.
2. Стивен Прата Язык программирования С++.
3. Дудзяний І.М. Програмування мовою С++.
4. Бьерн Страуструп Язык программирования С++.