|  |  |
| --- | --- |
| **Gerb-BMSTU_01** | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ **ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ**

КАФЕДРА **КОМПЬЮТЕРНЫЕ СИСТЕМЫ И СЕТИ (ИУ6)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.01 Информатика и вычислительная техника**

**ОТЧЕТ**

|  |  |
| --- | --- |
| **По лабораторной работе №3** |  |

**Название:** Оценка эффективности и качества программ.

**Дисциплина:** Технологии разработки программных систем.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент | ИУ-42б |  |  | С.В. Астахов |
|  | (Группа) |  | (Подпись, дата) | (И.О. Фамилия) |
|  |  |  |  |  |
| Преподаватель |  |  |  |  |
|  |  |  | (Подпись, дата) | (И.О. Фамилия) |

Москва, 2021

3 вариант

Введение

В настоящее время перед разработчиками программного обеспечения стоит задача создания эффективных, технологичных и качественных программ. Данная задача усложняется тем, что четких и универсальных рекомендаций для оценки указанных свойств не существует. Однако, на данный момент накоплен некоторый опыт, который может быть использован разработчиками.

**Цель работы:** изучить основные критерии оценки и способы повышения эффективности и качества программных продуктов.

Задача

Написать программу, которая генерирует массив уникальных случайных чисел в диапазоне [-100;+100]

Исходный код программы:

#include <stdio.h>

#include <stdlib.h>

#include <malloc.h>

#include <time.h>

int main()

{

int N = 10, i,j,k=0,f = 0;

double \*a;

double pr = 1;

a = (double\*)malloc(N \* sizeof(int));

srand(time(NULL));

a[0] = - 100 + rand()%(100 + 100 + 1);

for (i=1; i<N; i++){

while (f == 0){

a[i] = - 100 + rand()%(100 + 100 + 1);

for (j =0; j<i;j++){

if (a[i] != a[j]) k++;

}

if (k == i) f = 1;

k=0;

}

f = 0;

}

printf("Исходный массив:\n");

for (i=0; i<N; i++){

printf("%f\n ", a[i]);

}

for (i=0; i<N; i++){

pr \*= a[i];

}

printf("Произведение:%f\n",pr);

return 0;

}

Исходная программа работает некорректно, выдавая ошибки, так как при выделении памяти под массив адресуется память под N элементов типа int (4 байта на элемент), а записываются в массив элементы типа double (8 байт на элемент), следовательно, в ходе цикла программа выходит за пределы выделенной памяти.

Исправим эту ошибку, выделив память под N элементов типа double.

a = (double\*)malloc(N \* sizeof(double));

Программа стала работать корректно. Вывод программы:

Default array:

-98.000000

-22.000000

32.000000

-97.000000

-79.000000

89.000000

26.000000

-6.000000

-10.000000

-50.000000

Product:-3670136101632000.000000

Не меняя кода, реализующего основную логику программы, напишем «обертку», позволяющую считать эффективность программы по времени и использованию памяти при выполнении ее 100 000 раз:

#include <chrono>

#include <windows.h>

#include <Psapi.h>

using namespace std;

using namespace std::chrono;

int main()

{

high\_resolution\_clock::time\_point t1 = high\_resolution\_clock::now();

PROCESS\_MEMORY\_COUNTERS memCounter1;

BOOL result = K32GetProcessMemoryInfo(GetCurrentProcess(), &memCounter1, sizeof(memCounter1));

**for (int cnt = 0; cnt < 100000; cnt++) {**

**// Исходная программа**

**}**

high\_resolution\_clock::time\_point t2 = high\_resolution\_clock::now();

duration<double> time\_span = t2 - t1;

**std::cout << "It took me " << time\_span.count() << " seconds.";**

std::cout << std::endl;

PROCESS\_MEMORY\_COUNTERS memCounter2;

result = K32GetProcessMemoryInfo(GetCurrentProcess(), &memCounter2, sizeof(memCounter2));

**std::cout << "Memory leak: " << memCounter2.WorkingSetSize - memCounter1.WorkingSetSize << std::endl;**

std::cout << "\n(press any key to exit)\n";

std::getchar();

return 0;

}

Проведем первичное измерение характеристик программы:

Затраченное время: 0.177567 секунд

Затраты памяти: 13 303 808 байт

Аналитические утечки памяти: 100 000 \* 8 \* 10 = 8 000 000 байт (неосвобожденная память под массив)

Повышение эффективности программы

Предпримем ряд шагов по повышению эффективности программы, измеряя после каждого шага ее характеристики на 100 000 итераций.

1) В ходе программы выделяется динамическая память по массив а:

a = (double\*)malloc(N \* sizeof(double));

Однако эта память потом не освобождается. Исправим это, добавив освобождение памяти в конец программы.

free(a);

Фрагмент усовершенствованной программы:

for (int cnt = 0; cnt < 100000; cnt++) { //100k by default

int N = 10, i, j, k = 0, f = 0;

double\* a;

double pr = 1;

a = (double\*)malloc(N \* sizeof(double));

srand(time(NULL));

a[0] = -100 + rand() % (100 + 100 + 1);

for (i = 1; i < N; i++) {

while (f == 0) {

a[i] = -100 + rand() % (100 + 100 + 1);

for (j = 0; j < i; j++) {

if (a[i] != a[j]) k++;

}

if (k == i) f = 1;

k = 0;

}

f = 0;

}

//printf("Default array:\n");

for (i = 0; i < N; i++) {

//printf("%f\n ", a[i]);

}

for (i = 0; i < N; i++) {

pr \*= a[i];

}

//printf("Product:%f\n", pr);

**free(a);**

}

Проведем измерение характеристик программы:

Затраченное время: 0.183581 секунд

Затраты памяти: 237 568 байт

Как видно, временные характеристики программы **незначительно** ухудшились, затраты памяти же уменьшились **в несколько раз**.

2) Оптимизируем проверку очередного числа на уникальность: вместо подсчета уникальных чисел, будем менять флаг если найдено хоть одно повторяющееся число.

Фрагмент усовершенствованной программы:

for (int cnt = 0; cnt < 100000; cnt++) { //100k by default

int N = 10, i, j, k = 0, f = 0;

double\* a;

double pr = 1;

a = (double\*)malloc(N \* sizeof(double));

srand(time(NULL));

a[0] = -100 + rand() % (100 + 100 + 1);

for (i = 1; i < N; i++) {

while (f == 0) {

**f = 1;**

**a[i] = -100 + rand() % (100 + 100 + 1);**

**for (j = 0; j < i; j++) {**

**if (a[i] == a[j]) f=0;**

**}**

}

f = 0;

}

//printf("Default array:\n");

for (i = 0; i < N; i++) {

//printf("%f\n ", a[i]);

}

for (i = 0; i < N; i++) {

pr \*= a[i];

}

//printf("Product:%f\n", pr);

free(a);

}

Проведем измерение характеристик программы:

Затраченное время: 0.156702 секунд

Затраты памяти: 233 472 байт

Как видно, временные характеристики программы **незначительно** улучшились.

Повышение универсальности и технологичности программы

1) Исходная программа не дает пользователю возможности задать длину массива. Исправим это (при этом будем проверять, что введенные данные — целое число в диапазоне [1; 99] ).

Фрагмент усовершенствованной программы:

int N, i, j, k = 0, f = 0;

int inputN;

double\* a;

double pr = 1;

srand(time(NULL));

**printf("\n%s\n", "Input length of array (<100)");**

**inputN = scanf\_s("%d", &N);**

**if (inputN == 1 && N>0 && N<100) {**

a = (double\*)malloc(N \* sizeof(double));

a[0] = -100 + rand() % (100 + 100 + 1);

for (i = 1; i < N; i++) {

while (f == 0) {

f = 1;

a[i] = -100 + rand() % (100 + 100 + 1);

for (j = 0; j < i; j++) {

if (a[i] == a[j]) f = 0;

}

}

f = 0;

}

printf("\nDefault array:\n");

for (i = 0; i < N; i++) {

printf("%f\n ", a[i]);

}

for (i = 0; i < N; i++) {

pr \*= a[i];

}

printf("\nProduct:%f\n", pr);

}

**else {**

**printf("\n%s\n", "Invalid length of array");**

**}**

free(a);

2) Программа не позволяет пользователю указать ширину интервала для генерации значений массива. Добавим возможность ввести ее с клавиатуры (при этом будем проверять, что введенные данные — целое число >0).

Так же занесем инициализацию a[0] в цикл с целью улучшения стиля программы.

Фрагмент усовершенствованной программы:

int N, halfRange, i, j, k = 0, f = 0;

int inputN, inputHr;

double\* a;

double pr = 1;

srand(time(NULL));

printf("\n%s\n", "Input length of array (<100)");

inputN = scanf\_s("%d", &N);

if (inputN == 1 && N > 0 && N < 100) {

a = (double\*)malloc(N \* sizeof(double));

**printf("\n%s\n", "Input half-range of number");**

**inputHr = scanf\_s("%d", &halfRange);**

**if (inputHr == 1 && halfRange > 0) {**

for (**i = 0**; i < N; i++) {

while (f == 0) {

f = 1;

a[i] = -halfRange + rand() % (2 \* halfRange + 1);

for (j = 0; j < i; j++) {

if (a[i] == a[j]) f = 0;

}

}

f = 0;

}

printf("\nDefault array:\n");

for (i = 0; i < N; i++) {

printf("%f\n ", a[i]);

}

for (i = 0; i < N; i++) {

pr \*= a[i];

}

printf("\nProduct:%f\n", pr);

}

**else {**

**printf("\n%s\n", "Invalid half-range of element");**

**}**

free(a);

}

else {

printf("\n%s\n", "Invalid length of array");

}

3) Числа с плавающей точкой не ограничиваются по числу знаков после запятой, сделаем форматный вывод таких чисел (с двумя знаками после запятой).

Фрагмент усовершенствованной программы:

int N, halfRange, i, j, k = 0, f = 0;

int inputN, inputHr;

double\* a;

double pr = 1;

srand(time(NULL));

printf("\n%s\n", "Input length of array (<100)");

inputN = scanf\_s("%d", &N);

if (inputN == 1 && N > 0 && N < 100) {

a = (double\*)malloc(N \* sizeof(double));

printf("\n%s\n", "Input half-range of number >=N");

inputHr = scanf\_s("%d", &halfRange);

if (inputHr == 1 && halfRange > 0 && halfRange >= N) {

for (i = 0; i < N; i++) {

while (f == 0) {

f = 1;

a[i] = -halfRange + rand() % (2 \* halfRange + 1);

for (j = 0; j < i; j++) {

if (a[i] == a[j]) f = 0;

}

}

f = 0;

}

printf("\nDefault array:\n");

for (i = 0; i < N; i++) {

**printf("%.2f\n ", a[i]);**

}

for (i = 0; i < N; i++) {

pr \*= a[i];

}

**printf("\nProduct:%.2f\n", pr);**

}

else {

printf("\n%s\n", "Invalid half-range of element");

}

free(a);

}

else {

printf("\n%s\n", "Invalid length of array");

}

Пример работы усовершенствованной программы:

Input length of array (<100)

10

Input half-range of number >=N

15

Default array:

4.00

-2.00

-4.00

-3.00

-10.00

-1.00

9.00

-5.00

13.00

5.00

Product:2808000.00

Пример работы усовершенствованной программы при некорректных входных данных:

Input length of array (<100)

10

Input half-range of number >=N

5

Invalid half-range of element

Таблица 1 — оценка эффективности

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Критерий* | *Исходная программа* | | *Усовершенствованная программа* | |
| *Комментарий* | *Количественная оценка* | *Комментарий* | *Количественная оценка* |
| *Время* | Используется неоптимальный цикл проверки уникальности | 0.177567 секунд на 100 000 итераций | Оптимизирован цикл проверки уникальности | 0.156702 секунд на 100 000 итераций |
| *Оперативная память (аналитические подсчеты)* | Память, выделенная под массив, **не** осовобождается | 80 байт утечки  за 1 итерацию  8 000 000 байт утечки  за 100 000 итераций | Память, выделенная под массив, осовобождается | 0 байт утечки |
| *Оперативная память (результаты измерений* ***для процесса****)* | 13 303 808 байт  за 100 000 итераций | 233 472 байт  за 100 000 итераций |
| *Внешняя память* | Не используется | - | Не используется | - |

Таблица 2 — оценка качества программы

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Оценка* | *Правильность* | *Универсальность* | *Проверяемость* | *Точность результатов* |
| *Исходная программа* | Программа работает с ошибками из-за некорректного выделения памяти под массив | Программа работает с действительными числами, но не дает вводить длину массива и интервал случайных значений | Программа выводит все данные, ее можно проверить | Программа считает до 6 знаков после запятой, что избыточно в условиях данной задачи |
| *оценка* | 0/5 | 3/5 | 5/5 | 4/5 |
|  | Ошибка выделения памяти исправлена — программа работает корректно | Программа работает с действительными числами, и позволяет вводить длину массива и интервал случайных значений | Программа выводит все данные, ее можно проверить | Программа использует ту же точность для внутренних расчетов, но для удобства выводит числа в консоль до 2 знаков после запятой |
| *оценка* | - | - | - | - |

Вывод: в ходе данной работы были изучены методы оценки и повышения эффективности и качества программ.