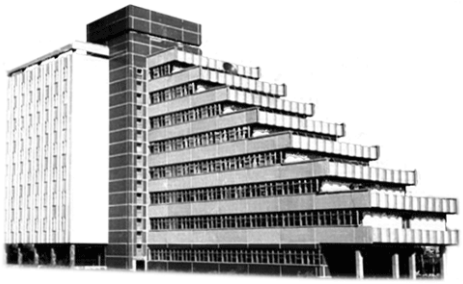
МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННО БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Учебный Центр Информационных Технологий «Информатика»



Лабораторная работа № 3
по дисциплине «Информатика и программирование 2 часть»

Выполнил слушатель: Пешков Е.В.
Вариант: 13
Дата сдачи:
Преподаватель: Юшманов А.А.

Новосибирск, 2019г.

1. Цель

Разработать две функции, одна из которых вводит с клавиатуры набор данных в произвольной последовательности и размещает в памяти в заданном формате. Другая функция читает эти данные и выводит на экран. Программа запрашивает и размещает в памяти несколько наборов данных при помощи первой функции, а затем читает их и выводит на экран при помощи второй. Размещение данных производить в выделенном массиве байтов с контролем его переполнения.

2. Вариант задания

Вариант 13

Область памяти представляет собой строку. Если в ней встречается выражение "%nnnd", где nnn - целое, то сразу же за ним следует массив из nnn целых чисел (во внутреннем представлении, то есть типа int). За выражением "%d" - одно целое число, за "%nnnf" - массив из nnn вещественных чисел, за "%f" - одно вещественное число.

3. Теория

Работа с памятью на низком уровне.

Операции преобразования типа указателя и адресной арифметики дают Си невиданную для языков высокого уровня свободу действий по управлению памятью. Традиционно языки программирования, даже если они работают с указателями или с их неявными эквивалентами - ссылками, не могут выйти за пределы единожды определенных типов данных для используемых в программе переменных. Напротив, в Си имеется возможность работать с памятью на «низком» уровне (можно сказать, ассемблерном или архитектурном). На этом уровне программист имеет дело не с переменными, а с помеченными областями памяти, внутри которых он может размещать данные любых типов и в любой последовательности, в какой только пожелает. Естественно, что при этом ответственность за корректность размещения данных ложится целиком на программиста.

Простейшим примером перехода от низкоуровневого представления к традиционному является использование функций динамического распределения памяти. Функция malloc возвращает указатель типа void* как адрес выделенной области памяти, который сразу же приводится к требуемому типа double*. Если исходить из заданной физической размерности области памяти (в байтах), то размерность полученного массива «промеряется» с использованием операции sizeof.

```
double   *d; int N;
printf (“Сколько байтов:”); scanf(“%d”,&N);
d=(double*)malloc(N);
int sz = N / sizeof(double);          // Количество вещественных в массиве
for (i=0; i < sz; i++) d[i] = i;
```

Операции преобразования типа указателя позволяют хранить разнотипные данные в аналогичном виде, но уже непосредственно в памяти. Заметим, что реализовать форматное представление данных можно только программно, т.е. динамически, статических описаний типов данных и переменных здесь недостаточно.

Для работы с последовательностью данных разных типов можно также применить объединение (union,), которое, как известно, позволяет размещать все свои элементы в общей памяти (не «друг за другом», а «друг на друге»). Если элементами union являются указатели, то все они «сливаются в один», к которому можно обращаться за любым указуемым типом.

Следующая программа упаковывает массив вещественных чисел, «сворачивая» последовательности подряд идущих нулевых элементов. Формат упакованной последовательности:

· последовательность ненулевых элементов кодируется целым счетчиком (типа int), за которым следуют сами элементы;

· последовательность нулевых элементов кодируется отрицательным значением целого счетчика;
· нулевое значение целого счетчика обозначает конец последовательности;

Исходная и упакованная последовательности выглядят так: 2.2, 3.3, 4.4, 5.5, 0.0, 0.0, 0.0, 1.1, 2.2, 0.0, 0.0, 4.4 и 4, 2.2, 3.3, 4.4, 5.5, -3, 2, 1.1, 2.2, -2, 1, 4.4, 0.

В процессе упаковки требуется подсчитывать количество подряд идущих нулей. В выходной последовательности запоминается место расположения последнего счетчика - также в виде указателя. Смена счетчика происходит, если текущий и предыдущий элементы относятся к разным последовательностям (комбинации «нулевой - ненулевой» и наоборот). Для записи в последовательность ненулевых значений из вещественного массива используется явное преобразование типа указателя int* в double*.

```
//----- Упаковка массива с нулевыми элементами
 void pack(int *p, double v[], int n)
 { int *pcnt=p++;                          // Указатель на последний счетчик
 *pcnt=0;                                  // Обнулить последний счетчик
 for (int i=0; i<n; i++){                          // Смена счетчика
         if (i!=0 && (v[i]==0 && v[i-1]!=0) || v[i]!=0 && v[i-1]==0)
         { pcnt=p++; *pcnt=0; }            // Обнулить последний счетчик
         if (v[i]==0) (*pcnt)--;           // -1 к счетчику нулевых
else {
         (*pcnt)++ ;                 // +1 к счетчику ненулевых
//                 *((double*)p)++ = v[i];     // сохранить само значение
                   double *q=(double*)p; *q++=v[i];
                   p=(int*)q;  }}
 *p++ = 0;}
 //------ Распаковка массива с нулевыми элементами
 int unpack(int *p, double v[])
 { int i=0,cnt;
 while ((cnt= *p++)!=0)                    // Пока нет нулевого счетчика
           {
           if (cnt<0)                      // Последовательность нулей
           while(cnt++!=0) v[i++]=0;
           else                            // Ненулевые элементы
           while(cnt--!=0)                 // извлечь с преобразованием
           v[i++]=*((double*)p)++;         // типа указателя
           double *q=(double*)p; v[i++] = *q++;
           p=(int*)q; }
           } return i;}
```

Преобразование «целое-указатель» и работа с машинными адресами

В конечном счете, значением указателя является адрес - обычное машинное слово определенной размерности, чему в Си соответствует целая переменная. Поэтому в Си преобразования типа «указатель-целое» и «целое-указатель» понимаются как получение адреса памяти в виде целого числа и преобразование целого числа в адрес памяти, то есть как работа с реальными адресами памяти компьютера. Такие операции являются машинно-зависимыми, поскольку требуют знания некоторых особенностей:

· системы преобразования адресов компьютера, размерностей используемых указателей (int или long);
· распределения памяти транслятором и операционной системой;
· архитектуры компьютера, связанной с размещением в памяти специальных областей (например, видеопамять экрана).

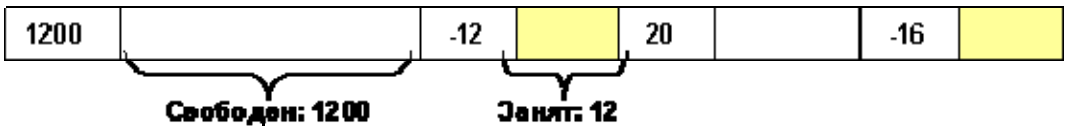
Естественно, что программа, использующая такие знания, не является переносимой (мобильной) и работает только в рамках определенного транслятора, операционной системы или компьютерной архитектуры. Типичный пример: если регистры устройства ввода-вывода размещаются в адресном пространстве основной (оперативной) памяти, то доступ к ним может быть реализован преобразованием константы-адреса к указателю, которое элементарно скрывается в define:

```
*(int*)0x1000=5;              // Запись целого 5 по шестнадцатеричному адресу 1000
#define REG *(char*)0xFFFFF0D0
REG=0xFF;                     // Запись константы «все 1» по адресу 0xFFFFF0D0
                              // регистра данных в пространстве адресов основной памяти
```

Динамическое распределение памяти

Как говорится, «не боги горшки обжигают». Система динамического распределения памяти (ДРП) может быть написана на самом же Си, но при этом обязательным будет сочетание работы с данными на низком и на высоком уровне. Внутреннее представление областей свободной и выделенной памяти может быть выполнено в виде любой динамической структуры данных (массив, массив указателей, список), работа с которой происходит на традиционном (высоком) уровне. Но размерности элементов этой структуры данных будут меняться, поэтому здесь не обойтись и без их физического (низкоуровневого) представления. К тому же управляющие компоненты структуры данных также являются динамическими, поэтому для них требуется динамическая память, распределяемая самой системой. Формальный парадокс: при использовании динамического массива указателей на свободные блоки во время выполнения функции free может произойти увеличение его размерности, что потребует вызова функций realloc или malloc/free той же самой проектируемой библиотеки.

В самом простом варианте система ДРП может быть реализована как последовательность занятых и свободных блоков, расположенных в памяти физически последовательно, т.е. друг за другом. В начале каждого блока размещается целая переменная , содержащая его размерность в байтах. Причем для обозначения занятости блока используется инвертированное (отрицательное) значение.



Для того, чтобы двигаться по цепочке блоков, можно использовать указатель типа char*, единица адресации которого соответствует одному байту. Тогда для извлечения счетчиков длин блоков необходимо преобразовать его «на лету» к int*. В исходном состоянии распределяемая область памяти представляет собой один свободный блок: в его начало записывается исходная длина.

```
// Система динамического распределения памяти
char *pa;                          // Область распределяемой памяти – «кучи»
int sz0;                           // Исходная размерность кучи
void create(int sz){               // Начальное состояние – один свободный блок
        pa=new char[sz];
        *(int*)pa=sz-sizeof(int);  // Размерность свободного куска – записать в начало
        sz0=sz; }
//-------------------------------------------------------------------
void _show(){                      // Просмотр состояния кучи
char *p=pa;
int lnt;
while(p<pa+sz0){                   // Пока не достигли адреса конца области
        lnt=*(int*)p;              // Извлечь из-под указателя длину блока
        if (lnt<0){                // Занятый блок - пропустить
```

```
                lnt=-lnt;           // Инвертировать длину
                printf("busy:");
                }
        else printf("free:");       // Вывести адрес (шестнадцатеричный) и длину
        printf(" addr=%8x sz=%d\n",p,lnt);
        p+=lnt+sizeof(int);         // Сдвинуть указатель на длину блока + длина счетчика
                }}
```

При выделении памяти имеется несколько стратегий поиска свободного блока. Наиболее простая из них: ищется свободный блок, строго подходящий по размеру. Если он обнаруживается, то отмечается как занятый. В противном случае необходимый блок «отрезается» от конца самого первого. При таком подходе самый первый блок будет всегда свободен, и от него будут отрезаться «недостающие» куски. Заметим, что такая стратегия хороша, когда преобладают запросы на выделение памяти блоками одного и того же размера.

```
// Поиск строго подходящего или отрезание от первого
void *_malloc(int sz){
char *p=pa;
int lnt;
while(p<pa+sz0){                        // Пока не достигли адреса конца области
        lnt=*(int*)p;                   // Извлечь из-под указателя длину блока
        if (lnt<0)                      // Занятый блок - пропустить
                p+=-lnt+sizeof(int);
        else    {
                if (sz==lnt) {          // Свободный – строго подходящий
                *(int*)p=-lnt;          // Обозначить как занятый
                return p+sizeof(int);   // Возвратить указатель на область данных
                }
                p+=lnt+sizeof(int);     // К следующему блоку
                }
        }
lnt=*(int*)pa;                          // Отрезать от первого – взять длину
if (sz+sizeof(int)>lnt) return NULL;    // Остаток мал – нет памяти
lnt -=sz+sizeof(int);                   // Уменьшить размер первого блока
*(int*)pa=lnt;                          // и записать полученный остаток
p=pa+lnt+sizeof(int);                   // Указатель на новый блок
*(int*)p=-sz;                           // Записать в него размерность (занят – <0)
return p+sizeof(int);                   // Возвратить указатель на память
}                                       // «вслед за» счетчиком
```

Функция выделения памяти возвращает указатель на память «вслед за» счетчиком длины выделенной области. Таким образом, ДРП запоминает, сколько памяти было выделено, и при выполнении функции освобождения первым делом смещает указатель назад к счетчику длины. Заметим, что данная версия ДРП в состоянии проверить корректность возвращаемого main-ом адреса занятого блока (хотя и не обязана это делать).

Кроме формального объявления блока свободным (путем инвертирования счетчика) необходимо еще «склеить» с текущим блоком предыдущий и последующий, если они также свободны (дефрагментация). Для поиска предыдущего блока приходится сделать лишний цикл просмотра содержимого распределяемой памяти с запоминанием указателя на предыдущий блок. Сама процедура «склеивания» заключается в увеличении длины предыдущего блока на размер текущего (+ счетчик длины текущего). Аналогичное действие предпринимается и для следующего блока

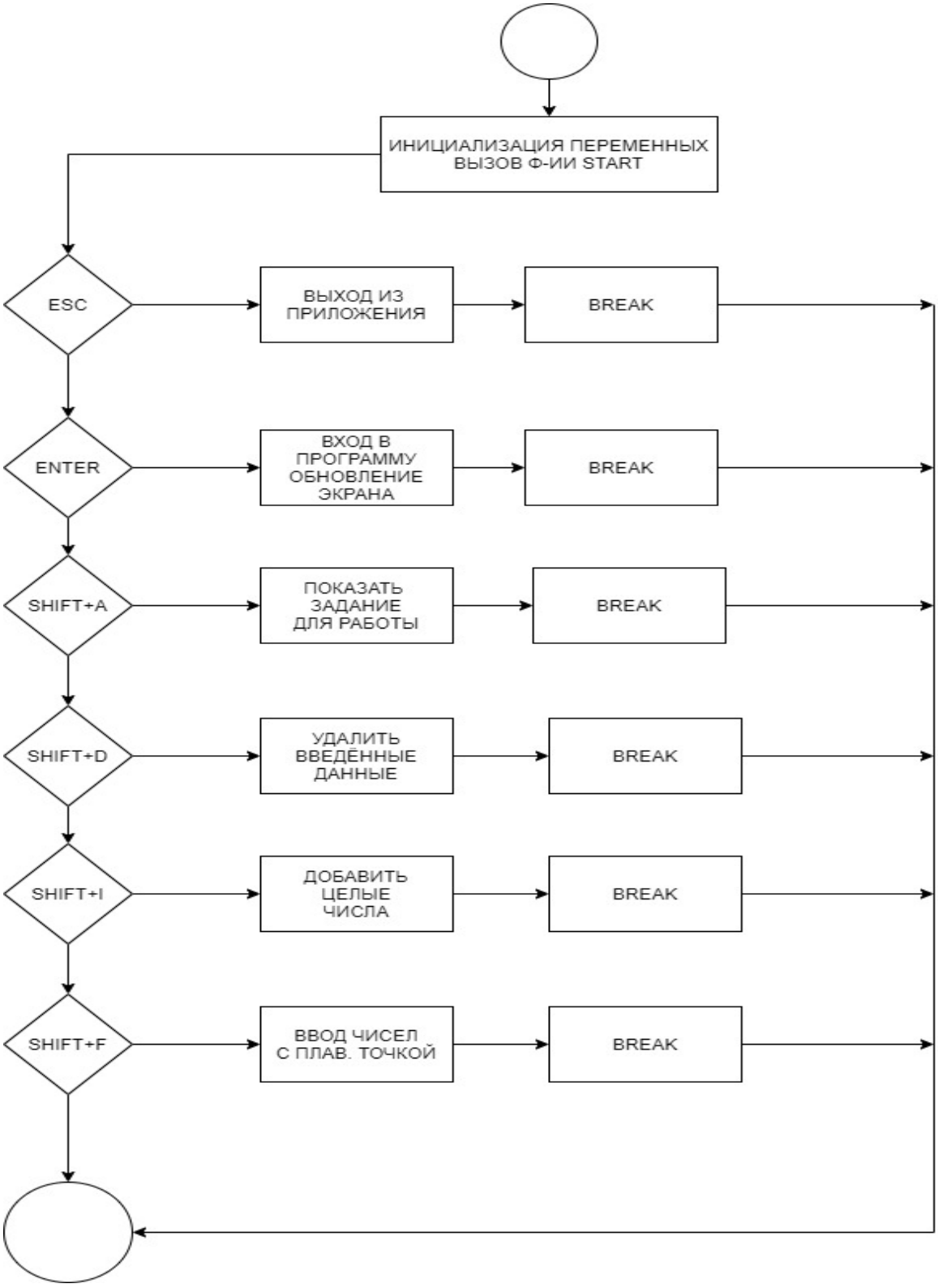
4. Анализ задачи и алгоритм

1) Анализ задачи

Входные данные: пустая строка заданного размера.

Результат: в заданной строке размещаем последовательно данные разного типа отделённые маркерами определённого вида, чтение (распаковка) выполняется при помощи операций приведения типа указателя.

2) Алгоритм решения задачи



5. Описание программной реализации

1) Используемые переменные

char string[BUFFER_SIZE] – строка для размещения данных
char* curent_point = string – указатель на текущую позицию в строке

2) Используемые функции

Функция char* input_int_numbers(char* curent_point, int free_mem);
Аргументы функции:
char* curent_point – позиция начала записи.
int free_mem – количество свободных байт
Возвращаемый результат: указатель на начало позиции для следующей записи.
Принцип работы: записывает целые числа в строку.

Функция char* input_float_numbers(char* curent_point, int free_mem);
Аргументы функции:
char* curent_point – позиция начала записи.
int free_mem – количество свободных байт
Возвращаемый результат: указатель на начало позиции для следующей записи.
Принцип работы: записывает вещественные числа в строку.

Функция int overflow_control(int free_mem, int count_numbers);
Аргументы функции:
int free_mem – количество свободной памяти.
int count_numbers – количество чисел для размещения
Возвращаемый результат: 1 если памяти недостаточно 0 если достаточно.
Принцип работы: контролирует переполнение памяти.

Функция void clear_data(char* start_point, int total);
Аргументы функции:
char* start_point – указатель на начало строки.
int total – длинна строки
Возвращаемый результат: ничего не возвращает.
Принцип работы: обнуляет строку.

Функция void string_unpacking(char* start_point);
Аргументы функции:
char* start_point – указатель на начало строки.

Возвращаемый результат: ничего не возвращает.
Принцип работы: распаковывает записанные данные.

Функция void string_view(char* start_point);
Аргументы функции:
char* start_point – указатель на начало строки.

Возвращаемый результат: ничего не возвращает.
Принцип работы: выводит в консоль строковое представление упакованных данных.

Функция void info();
Аргументы функции:

Возвращаемый результат: ничего не возвращает.
Принцип работы: выводит в консоль информацию о задании для лабораторной.

Функция void print_header();
Аргументы функции:

Возвращаемый результат: ничего не возвращает.
Принцип работы: выводит в консоль заголовок с командами для работы с программой.

Функция void print_footer(int total, int free_mem);
Аргументы функции:
int total – общее количество выделенной памяти
int free_mem - количество свободных байт
Возвращаемый результат: ничего не возвращает.
Принцип работы: выводит в консоль информацию о памяти выделенной для работы с программой.

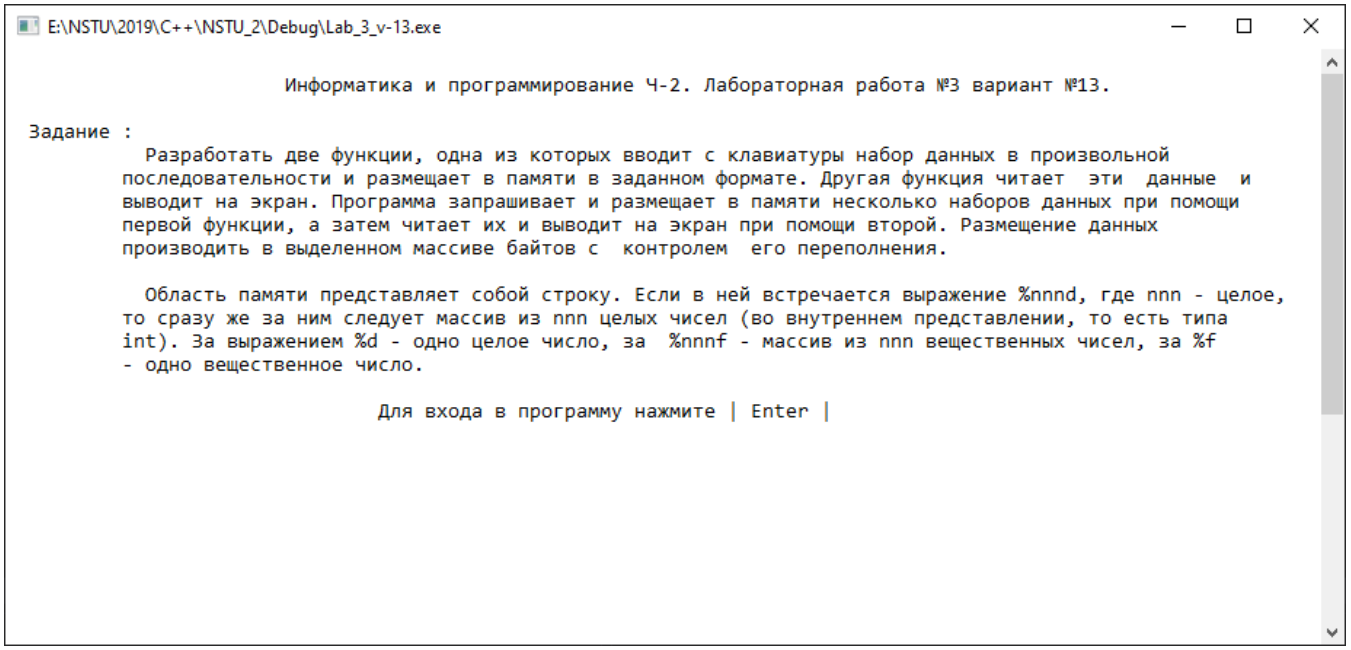
Функция void print_line();
Аргументы функции:

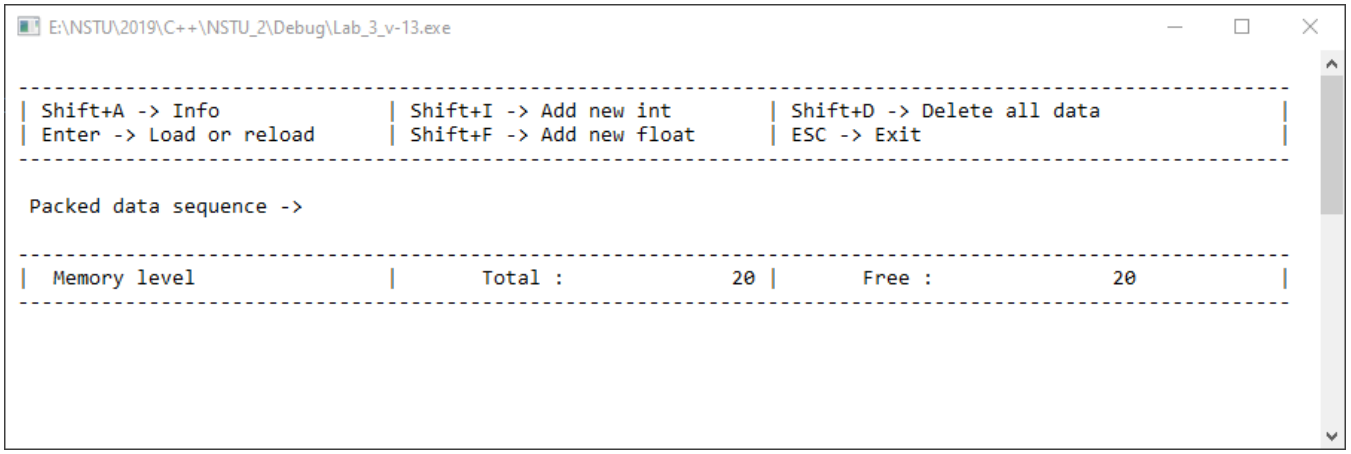
Возвращаемый результат: ничего не возвращает.
Принцип работы: выводит в консоль разделительную линию.

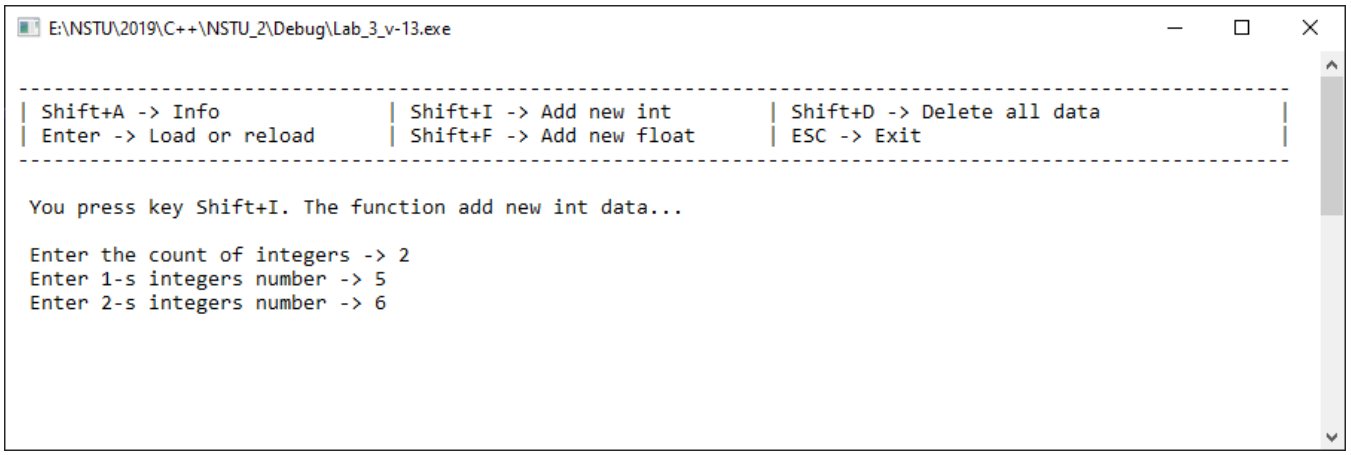
Функция void start(char* start_point, char* curent_point, int total);
Аргументы функции:
char* start_point – указатель на начало строки.
char* curent_point – указатель на текущую позицию в строке
int total – длинна строки

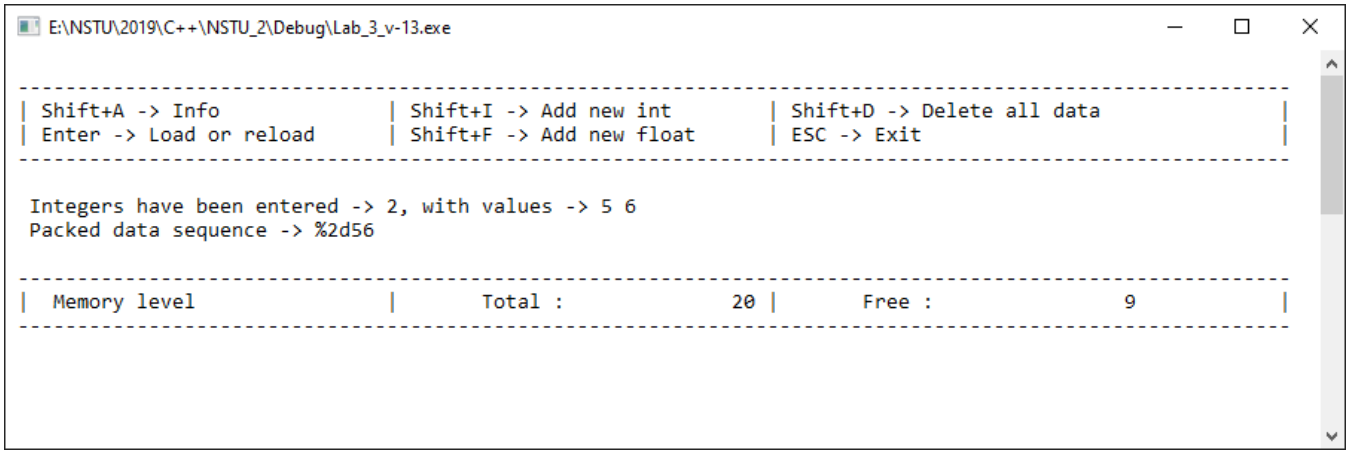
Возвращаемый результат: ничего не возвращает.
Принцип работы: запускает программу обрабатывает нажатие клавиш.

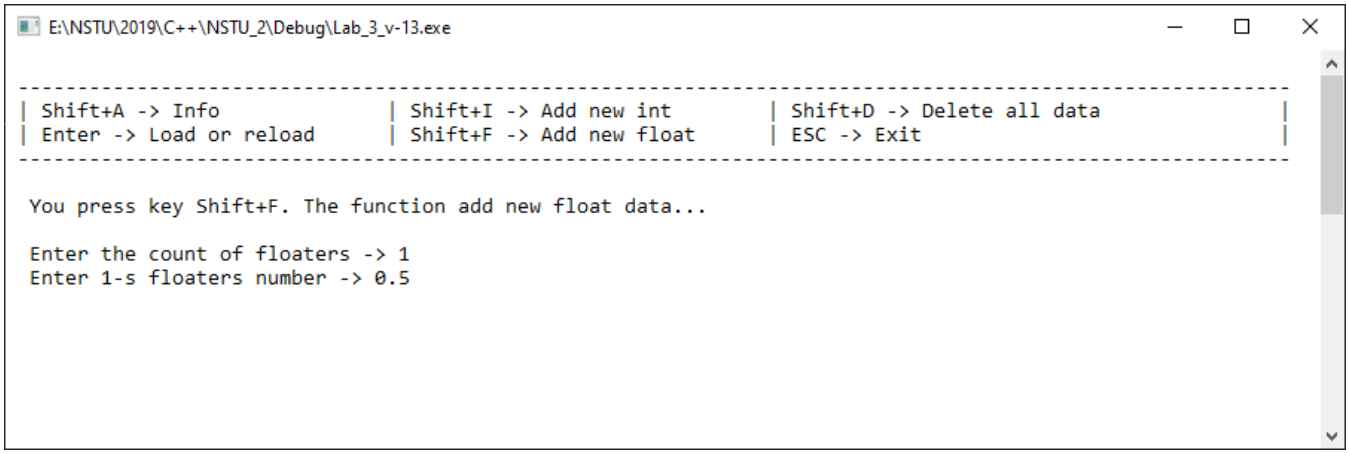
6. Пример работы программы

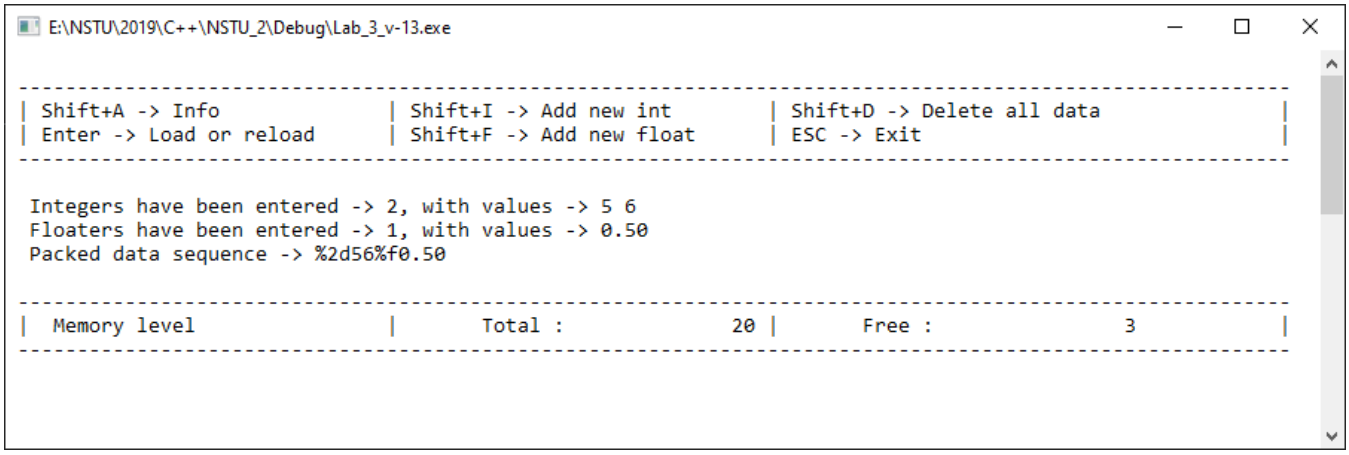












7. Выводы

В ходе выполнения лабораторной работы были изучены принципы размещения данных разного типа в выделенном массиве байт.

Приложение. Текст программы

Файл header.h

```
#pragma once
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <stdbool.h>
#include <limits.h>
#include <float.h>

#define BUFFER_SIZE 20

#define ENTER 13
#define ESC 27
#define SHIFT_A 65
#define SHIFT_D 68
#define SHIFT_I 73
#define SHIFT_F 70

char* input_int_numbers(char* curent_point, int free_mem);
char* input_float_numbers(char* curent_point, int free_mem);
int overflow_control(int free_mem, int count_numbers);
void clear_data(char* start_point, int total);

void string_unpacking(char* start_point);
void string_view(char* start_point);

void info();
void print_header();
void print_footer(int total, int free_mem);
void print_line();
void start(char* start_point, char* curent_point, int total);
```

Файл main.c

```
#include "header.h"

int main()
{
	system("chcp 1251");
	system("mode 115,40");

	char string[BUFFER_SIZE];
	char* curent_point = string;

	int total = sizeof(string) / sizeof(string[0]);

	start(string, curent_point, total);

	return 0;
}
```

Файл menu.c

```
#include "header.h"

void print_header()
{
   printf("\n");
   print_line();
   printf(" | Shift+A -> Info \t\t| Shift+I -> Add new int  \t| Shift+D -> Delete all data\t\t
|\n");
   printf(" | Enter -> Load or reload\t| Shift+F -> Add new float\t| ESC -> Exit\t\t\t\t   |\n");
   print_line();
   printf("\n");
}

void print_footer(int total, int free_mem)
{
   printf("\n\n");
   print_line();
   printf(" |  Memory level\t\t|\tTotal :%16d |\tFree :%17d\t\t   |\n", total, free_mem);
   print_line();
   printf("\n");
}

void print_line()
{
   printf(" ");
   for (int i = 0; i < 107; i++) {
          printf("-");
   }
   printf("\n");
}

void info()
{
   printf("\n\t\t\tИнформатика и программирование Ч-2. Лабораторная работа №3 вариант №13.\n\n");
   printf("  Задание : \n");
   printf("\t    Разработать две функции, одна из которых вводит с клавиатуры набор данных в произ-
вольной\n");
   printf("\t  последовательности и размещает в памяти в заданном формате. Другая функция читает
эти  данные  и\n");
   printf("\t  выводит на экран. Программа запрашивает и размещает в памяти несколько наборов
данных при помощи\n");
   printf("\t  первой функции, а затем читает их и выводит на экран при помощи второй. Размещение
данных \n");
   printf("\t  производить в выделенном массиве байтов с  контролем  его переполнения. \n\n");
   printf("\t    Область памяти представляет собой строку. Если в ней встречается выражение %%nnnd,
где nnn - целое, \n");
   printf("\t  то сразу же за ним следует массив из nnn целых чисел (во внутреннем представлении,
то есть типа \n");
   printf("\t  int). За выражением %%d - одно целое число, за  %%nnnf - массив из nnn вещественных
чисел, за %%f \n");
   printf("\t  - одно вещественное число. \n\n");
   printf("\t\t\t\tДля входа в программу нажмите | Enter |\n");
}

void start(char* start_point, char* curent_point, int total)
{
   system("cls");
   info();
   curent_point = start_point;
   int free_mem;
   while (true)
   {
          switch (_getch())
          {
          case ESC:
```

```
                exit(0);
                break;

        case ENTER:
                system("cls");
                print_header();
                string_unpacking(start_point);
                string_view(start_point);
                free_mem = total - (curent_point - start_point);
                print_footer(total, free_mem);
                break;

        case SHIFT_A:
                system("cls");
                info();
                break;

        case SHIFT_F:
                system("cls");
                print_header();
                printf("  You press key Shift+F. The function add new float data...\n\n");
                free_mem = total - (curent_point - start_point);
                curent_point = input_float_numbers(curent_point, free_mem);
                break;

        case SHIFT_I:
                system("cls");
                print_header();
                printf("  You press key Shift+I. The function add new int data...\n\n");
                free_mem = total - (curent_point - start_point);
                curent_point = input_int_numbers(curent_point, free_mem);
                break;

        case SHIFT_D:
                system("cls");
                print_header();
                clear_data(start_point, total);
                string_unpacking(start_point);
                string_view(start_point);
                curent_point = start_point;
                free_mem = total - (curent_point - start_point);
                print_footer(total, free_mem);
                break;
        }
   }
}
```

Файл pack.c

```
#include "header.h"

char* input_int_numbers(char* curent_point, int free_mem)
{
	int count_numbers;
	printf("  Enter the count of integers -> ");
	scanf("%d", &count_numbers);

	int break_flag = overflow_control(free_mem, count_numbers);
	if (break_flag == 1) {
		printf("  Not enough memory to write!!!");
		return curent_point;
	}
	else {
		*curent_point++ = '%';
		if (count_numbers > 1)
		{
			*curent_point++ = (char)count_numbers;
		}
		*curent_point++ = 'd';

		int integer_int;
		for (int i = 0; i < count_numbers; i++) {
			do
			{
				printf("  Enter %d-s integers number -> ", i + 1);
				scanf("%d", &integer_int);
				if (integer_int < INT_MIN || integer_int > INT_MAX)
				{
					printf("  The entered number is out of range !");
				}
			} while (integer_int < INT_MIN || integer_int > INT_MAX);
			*((int*)curent_point)++ = integer_int;
		}
	}
	return curent_point;
}

char* input_float_numbers(char* curent_point, int free_mem)
{
	int count_numbers;
	printf("  Enter the count of floaters -> ");
	scanf("%d", &count_numbers);

	int break_flag = overflow_control(free_mem, count_numbers);
	if (break_flag == 1) {
		printf("  Not enough memory to write!!!");
		return curent_point;
	}
	else {
		*curent_point++ = '%';
		if (count_numbers > 1)
		{
			*curent_point++ = (char)count_numbers;
		}
		*curent_point++ = 'f';

		float float_f;
		for (int i = 0; i < count_numbers; i++) {
			do {
				printf("  Enter %d-s floaters number -> ", i + 1);
				scanf("%f", &float_f);
				if (float_f < FLT_MIN || float_f > FLT_MAX)
				{
					printf("  The entered number is out of range !");
```

```
                        }
                    } while (float_f < FLT_MIN || float_f > FLT_MAX);
                    *((float*)curent_point)++ = float_f;
                }
        }
        return curent_point;
}

int overflow_control(int free_mem, int count_numbers)
{
        int size_marker = count_numbers > 1 ?  3 :  2;
        if (free_mem < ((sizeof(count_numbers) * count_numbers) + size_marker))
        {
                return 1;
        }
        return 0;
}

void clear_data(char* start_point, int total)
{
        for (int i = 0; i < total; i++)
        {
                *start_point++ = 0;
        }
}
```

Файл upack.c

```
#include "header.h"

void string_unpacking(char* start_point)
{
        int count_numbers;

        while (*start_point++ == '%') {
                if (*start_point == 'd' || *start_point == 'f') {
                        count_numbers = 1;
                }
                else {
                        count_numbers = (int)*start_point++;
                }
                if (*start_point++ == 'd') {
                        printf("  Integers have been entered -> %d, with values -> ",
count_numbers);
                        int integer_int;
                        for (int i = 0; i < count_numbers; i++) {
                                integer_int = *((int*)start_point)++;
                                printf("%d ", integer_int);
                        }
                        printf("\n");
                }
                else {
                        printf("  Floaters have been entered -> %d, with values -> ",
count_numbers);
                        float float_f;
                        for (int i = 0; i < count_numbers; i++) {
                                float_f = *((float*)start_point)++;
                                printf("%.2f ", float_f);
                        }
                        printf("\n");
                }
        }
}
```

```
void string_view(char* start_point)
{
	int count_numbers;

	printf("  Packed data sequence -> ");
	while (*start_point++ == '%') {
		if (*start_point == 'd' || *start_point == 'f') {
			count_numbers = 1;
		}
		else {
			count_numbers = (int)*start_point++;
		}
		if (*start_point++ == 'd')
		{
			printf("%%");
			if (count_numbers == 1) {
				printf("d");
			}
			else {
				printf("%dd", count_numbers);
			}
			int integer_int;
			for (int i = 0; i < count_numbers; i++) {
				integer_int = *((int*)start_point)++;
				printf("%d", integer_int);
			}
		}
		else
		{
			printf("%%");
			if (count_numbers == 1) {
				printf("f");
			}
			else {
				printf("%df", count_numbers);
			}
			float float_f;
			for (int i = 0; i < count_numbers; i++) {
				float_f = *((float*)start_point)++;
				printf("%.2f", float_f);
			}
		}
	}
}
```