# Лабораторная работа №8

Способы вызова ассемблерных подпрограмм

в языках высокого уровня

**Цель работы**: изучение способов вызова подпрограмм, написанных на разных языках программирования посредством dll-библиотек.

**Теоретические сведения**

Подпрограммы, написанные на каком-либо языке программирования можно компоновать в динамические библиотеки (dll-библиотеки), которые представляют собой скомпилированный объектный код, и подключать к другим программам. dll-библиотека содержит в себе набор процедур, функций, а также других ресурсов, которые могут быть использованы программой, которые её подключают. После подключения библиотеки, основная программа может вызывать из неё подпрограммы, если известны их спецификации, т.е. количество аргументов, их тип и тип возвращаемого значения. Заранее спецификации могут быть не известны, т.к. как dll-библиотека содержит объектный код и для подпрограмм известны только их имена. В отличие от библиотек языков высокого уровня, например файлов с расширением \*.h языка c++, или файлов \*.cs языка C#, которые представляют собой исходный код, dll-библиотека представляет собой уже скомпилированный модуль с известными точками (адресами) входа в подпрограммы.

Для того, чтобы корректно вызывать подпрограммы, написанные на разных языках программирования, существуют правила, по которым необходимо передавать аргументы, возвращать данные и очищать стек. Такие правила называются *соглашениями о вызовах*.

Рассмотрим пример простой dll-библиотеки на ассемблере, которая содержит инициализирующую функцию DllMain, функцию sum\_int\_s для сложения чисел типа int, функцию mul\_double умножения чисел типа double, функцию memcopy32 копирования блока памяти, размер которого кратен 4 байтам. Все подпрограммы в ней реализованы в стиле вызова stdcall, который указан в начале библиотеки.

Соответствие типов языков C#, С++ директивам объявления   
данных на ассемблере.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| C# | C++ | Размер (байт) | Ассемблер | Место возврата |
| int, uint | int, unsigned int, long | 4 | dd (DWORD) | EAX |
| byte | char, unsigned char | 1 | db (BYTE) | AL |
| float | float | 4 | dd (DWORD) | ST(0) |
| double | double, long double | 8 | dq (QWORD) | ST(0) |
| short, ushort | short, unsigned short | 2 | dw (WORD) | AX |
| bool | bool | 1 | db (BYTE) | AL |
| int[], float[],…  (любой указатель) | int \*, float \*, …  (любой указатель) | 4 | dd (DWORD) | EAX |
| int, uint | long | 4 | dd (DWORD) | EAX |
| long, ulong | long long | 8 | dq (QWORD) | EDX:EAX |

.386

.model flat, stdcall

option casemap :none

.code

; Инициализирующая функция

**DllMain proc** hlnstDLL:DWORD, reason:DWORD, unused:DWORD

mov EAX, 1

ret

**DllMain Endp**

; Функция возвращает сумму чисел a и b типа int

**sum\_int\_s proc**

mov EAX, [ESP+4] ; Здесь находится a

add EAX, [ESP+8] ; Здесь находится b

ret 8

**sum\_int\_s endp**

; Функция возвращает произведение чисел a и b типа double

**mul\_double proc**

push EBP ; Сохраняем EBP в стеке

mov EBP, ESP ; EBP = ESP

; Смещаем указатель относительно вершины стека на 8, чтобы EBP указывал на первый аргумент a

add EBP, 8

finit

fld QWORD PTR [EBP] ; Загрузить a в ST(0)

fld QWORD PTR [EBP+8] ; Загрузить b в ST(0)

fmul ; ST(0) = ST(0) \* ST(1)

pop EBP

ret 16

**mul\_double endp**

; Процедура копирования блока памяти, размер которого кратен 4 байтам

; memcopy32 (void\* dest, void\* source, int mem\_size)

**memcopy32 proc**

; Сохранение используемых регистров в стеке

push ESI

push EDI

push ECX

mov ecx, [ESP+24] ; ECX = mem\_size

shr ecx, 2 ; ECX = ECX / 4 (mem\_size = mem\_size >> 2)

cld ; Очистка флага направления

mov esi, [ESP+20] ; ESI = source

mov edi, [ESP+16] ; EDI = dest

rep movsd ; Копирование блока

; Восстановление используемых регистров

pop ECX

pop EDI

pop ESI

ret 12

**memcopy32 endp**

**sum\_int\_s2** proc a:DWORD, b:DWORD

mov EAX, a

add EAX, b

ret

**sum\_int\_s2** endp

End LibMain

dll-библиотека должна содержать инициализирующую функцию (DllMain), которая срабатывает при загрузке библиотеки и в некоторых случаях вызывается операционной системой. В ней можно выполнять проверку и инициализацию необходимых ресурсов. Данная функция должна возвращать истину (-1), если подключение прошло успешно и можно продолжать работу, или ложь (0) для завершения работы.

Функция sum\_int\_s2, в отличие от других, содержит в своём заголовке список аргументов, к которым можно обращаться по имени и не вычислять их смещение относительно вершины стека. Компилятор добавил для sum\_int\_s2 некоторую последовательность команд в начало и в конец подпрограммы. Последовательность добавляемых команд в начале подпрограммы и перед её вызовом называется *прологом*, в конце и после её вызова – *эпилогом*. В отладчике можно увидеть, что функция sum\_int\_s2, в которую добавлен пролог и эпилог, содержит следующие команды:

PUSH EBP ; Пролог

MOV EBP,ESP ; Пролог

MOV EAX,DWORD PTR [EBP+8] ; mov EAX, a

ADD EAX,DWORD PTR [EBP+0C] ; add EAX, b

LEAVE ; Эпилог

RET 8 ; ret, эпилог

Первые две команды являются прологом и сохраняют содержимое EBP в стеке, и копируют значение **ESP** в регистр EBP. Последние две команды являются эпилогом. Команда **LEAVE**, наоборот, копирует содержимое **EBP** в **ESP** и затем извлекает из стека 4 байта и помещает их в регистр **EBP**. Таким образом, если содержимое EBP нигде в подпрограмме не изменяется, удобно обращаться к аргументам в стеке по смещению относительно адреса, который содержится в **EBP**, и свободно пользоваться стеком. При этом замену имени аргумента компилятор выполняет сам. Т.е., если в исходной программе есть команда «mov EAX, a», то компилятор автоматически вычислит местонахождение аргумента *a* (его смещение относительно вершины) в стеке и заменит исходную команду на «MOV EAX,DWORD PTR [EBP+8]». Разумеется, если пролог и эпилог включен, использовать регистр **EBP** в своих нуждах уже нельзя.

По умолчанию компилятор добавляет код пролога и эпилога в подпрограммы согласно стилю вызова, который объявлен в начале файла (.model flat, stdcall). Для разных стилей вызова код пролога и эпилога различен. Можно переопределить стиль для конкретной подпрограммы в её заголовке, указав его после директивы **proc**:

sum\_int\_s3 proc **stdcall** a:DWORD, b:DWORD ; Стиль stdcall

sum\_int\_c3 proc **c** a:DWORD, b:DWORD ; Стиль cdecl

sum\_int\_p3 proc **pascal** a:DWORD, b:DWORD ; Стиль pascal

Можно отключить автоматическую генерацию кода пролога и эпилога с помощью следующих команд:

OPTION PROLOGUE : NONE

OPTION EPILOGUE : NONE

Эти команды можно добавить, к примеру, после строки «.code». Если пролог и эпилог будет опять нужен в какой-то части программы, можно включить его генерацию командами:

OPTION PROLOGUE:PrologueDef

OPTION EPILOGUE:EpilogueDef

В вышеприведённой программе используется стиль вызова **stdcall**. В данном стиле вызова аргументы передаются в подпрограмму (помещаются в стек) справа налево и вызываемая подпрограмма должна сама очищать стек от аргументов. **stdcall** используется API-функциями ОС Windows.

В языке **Си** используется стиль вызова **cdecl** (от англ. c-declaration). Аргументы функции передаются в стек справа налево, стек очищает вызывающая программа. Для данного стиля вызова коды пролога и эпилога добавляются также в вызывающую подпрограмму. Можно явно указывать (переопределять) стиль вызова для отдельных подпрограмм. Добавим в библиотеку функцию сложения чисел в стиле **cdecl** с явным перечислением аргументов при включённой генерации пролога и эпилога.

**sum\_int\_c2** proc **c** a:DWORD, b:DWORD ; c – стиль cdecl

mov EAX, a

add EAX, b

ret

**sum\_int\_c2** endp

Стиль вызова здесь переопределён после директивы **proc**.

Если проанализировать код данной функции в отладчике, то можно увидеть, что при выходе из подпрограммы стек от аргументов ею не очищается, поэтому это нужно делать в основной подпрограмме после вызова:

PUSH 1 ; Загрузка аргумента b

PUSH 10 ; Загрузка аргумента a

CALL sum\_int\_c2

ADD ESP, 8 ; Очищение стека от аргументов

В данном случае команда «ADD ESP, 8» команда является частью эпилога. Та же подпрограмма, но без явного перечисления аргументов выглядит следующим образом:

sum\_int\_c proc

mov EAX, [ESP+4]

add EAX, [ESP+8]

ret

sum\_int\_c endp

Стиль вызова **pascal** используется в языке Паскаль. Аргументы передаются слева направо, стек очищает вызываемая подпрограмма.

Стиль вызова **fastcall** используется компиляторами Borland и подразумевает передачу аргументов через регистры и стек. Первые аргументы передаются через регистры, остальные – через стек. В компиляторах фирмы Microsoft данный стиль вызова также поддерживается. При этом первые два аргумента передаются через регистры **CL**/**CX**/**ECX** и **DL**/**DX**/**EDX**.

Компиляторами языка C++ при использовании объектно-ориентированного подхода используется стиль вызова **thiscall**. Указатель на объект (**this**) передаётся в регистр ECX, аргументы подпрограммы передаются в стек справа налево, стек очищает вызываемая подпрограмма.

В следующем таблице перечислены реализации функции сложения sum\_int целых чисел в различных стилях вызова.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Стиль  вызова | Тело функции | Вызов  sum\_int(7, 10) | Способ очистки  стека | Порядок передачи аргумен-тов |
| stdcall | mov EAX, [ESP+4]  add EAX, [ESP+8]  ret 8 | PUSH 10  PUSH 7  CALL sum\_int | Вызывае-мая | Справа налево |
| cdecl | mov EAX, [ESP+4]  add EAX, [ESP+8]  ret | PUSH 10  PUSH 7  CALL sum\_int  ADD ESP, 8 | Вызыва-ющая | Справа налево |
| fastcall | mov EAX, ECX  add EAX, EDX  ret | MOV EDX, 10  MOV ECX, 7  CALL sum\_int | Вызывае-мая | Справа налево |
| pascal | mov EAX, [ESP+8]  add EAX, [ESP+4]  ret 8 | PUSH 7  PUSH 10  CALL sum\_int | Вызывае-мая | Слева направо |

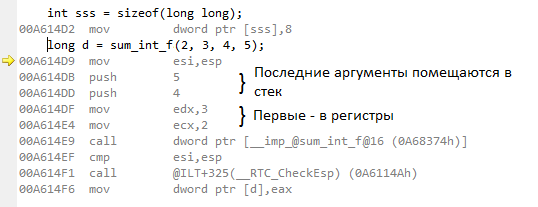
Чтобы легче было разобраться в способах вызова и упростить отладку программ, можно дизассемблировать участок кода в среде Visual Studio (пункт главного меню Debug->Windows->Disassembly) и посмотреть, в каком порядке компилятор ожидает аргументы, и кто, по его мнению, должен освобождать стек.

К примеру, пусть объявлена функция, которая имеет 4 аргумента:

extern "C" \_\_declspec(dllimport) int \_fastcall

sum\_int\_f(int a, int b, int c, int d);

Если выполнить отладку (в режиме Debug), то в окрестности её вызова наблюдается следующий код:



Ясно, что последние два аргумента помещаются в стек командами **push**, а первые два – через регистры. Значит функция, которая написана на ассемблере, должна быть реализована в соответствии c именно этим порядком. Далее вызывается функция, имя которой декорировано. Можно проанализировать и другие команды. Перед вызовом функции значение **ESP** копируется в **ESI**, а после вызова – проверяется командой **cmp**. По названию следующей функции \_\_RTC\_CheckESP можно догадаться, что она проверяет, не нарушена ли структура стека путём анализа флага **ZF**. Если подпрограмма правильно освободила стек, то **ESP** должен быть равен **ESI**. В последней команде переменной **d** присваивается значение EAX, т.е. возвращаемое значение. Отметим, что нигде нет команды, очищающей стек от аргументов (**ADD ESP, 16**), значит, программа на C++ ожидает, что стек очистит функция **sum\_int\_f**.

Для того чтобы отметить в dll-библиотеке и файле \*.lib, какой стиль вызова использует каждая подпрограмма и сколько она имеет аргументов применяют *декорирование* имён. При декорировании перед названием функции добавляется символ подчёркивания «\_» или «@», в зависимости от стиля вызова, и в конце после символа «@» добавляется число, указывающее суммарный размер аргументов в байтах, помещаемых в стек. Размер аргументов кратен 4 байтам. Компиляторами Microsoft/Intel C++ используется следующий способ декорирования:

|  |  |  |
| --- | --- | --- |
| \_cdecl | \_fastcall | \_stdcall |
| \_int\_sum | @int\_sum@8 | \_int\_sum@8 |

Если аргументов нет, то в конце добавляется «@0». Вышеприведённым функциям будут присвоены следующие имена:

\_sum\_int\_s@0

\_sum\_int\_с@0

\_mul\_double@0

\_memcopy32@0

\_sum\_int\_s2@8

\_sum\_int\_с2

Функция **sum\_int\_f** с заголовком без аргументов «**sum\_int\_f proc**» будет иметь имя \_sum\_int\_f@0. Для вызова первых 4 подпрограмм на языке C++ такие имена не годятся, так как они на самом деле имеют аргументы, поэтому нужно переопределить эти имена при сборке и указать правильно размер аргументов. В исходном коде аргументы у них не перечислены, поэтому компоновщик добавляет в конце «@0». Стиль вызова для всего модуля общий – stdcall, поэтому в начало имени добавляется символ подчёркивания «\_». Список имён всех функций, экспортируемых в dll-библиотеку, нужно записать в файл с расширением \*.def. Здесь же можно переопределить имена. Содержимое файла lab8.def:

LIBRARY lab8

EXPORTS

\_mul\_double@16 = \_mul\_double@0

\_memcopy32@12 = \_memcopy32@0

\_sum\_int\_s@8 = \_sum\_int\_s@0

sum\_int\_s2

\_sum\_int\_c = \_sum\_int\_c@0

sum\_int\_c2

@sum\_int\_f@16 = \_sum\_int\_f@0

Для функций sum\_int\_s2 и sum\_int\_c2 имена можно не переопределять, так как они декорированы верно.

Для компиляции dll-библиотеки с командной строки используются следующие ключи:

**D:\lab8>**D:\masm32\bin\ml /c /coff lab8.asm

**D:\lab8>**D:\masm32\bin\Link /SUBSYSTEM:WINDOWS /DLL /DEF:lab8.def lab8.obj

Если всё сделано правильно, то будут созданы файлы lab8.dll и lab8.lib.

Статическое подключение dll-библиотеки

Сначала нужно скопировать файлы lab8.lib и lab8.dll в каталог с проектом на C++, к которому они будут подключаться. Для подключения библиотеки в среде Visual Studio программе на C++ необходимо добавить в начале программы строку

#pragma comment(lib,"lab8.lib")

и описать заголовки функций:

extern "C" \_\_declspec(dllimport) void \_stdcall memcopy32(void\* dest, void\* source, int block\_size);

extern "C" \_\_declspec(dllimport) int \_stdcall sum\_int\_s (int a, int b);

extern "C" \_\_declspec(dllimport) int \_stdcall sum\_int\_s2(int a, int b);

extern "C" \_\_declspec(dllimport) int \_cdecl sum\_int\_c (int a, int b);

extern "C" \_\_declspec(dllimport) int \_cdecl sum\_int\_c2(int a, int b);

extern "C" \_\_declspec(dllimport) double \_stdcall mul\_double(double a, double b);

extern "C" \_\_declspec(dllimport) int \_fastcall sum\_int\_f(int a, int b, int c, int d);

extern "C" означает, что к именам функций будет применяться декорирование по правилам языка Си (описано выше). Атрибут \_\_declspec(dllimport) объявляет все функции как импортируемые извне, т.е. из внешней dll-библиотеки. Далее функции можно использовать для вызова.

На C++ можно не только подключать, но и создавать dll-библиотеки. В этом случае нужно будет применять атрибут \_\_declspec(dllexport) к именам, т.е. подпрограммы будут экспортироваться в библиотеку.

Для подключения библиотеки на языке C# нужно скопировать только файл \*.dll в каталог с исполняемым \*.exe файлом проекта и объявить подпрограммы следующим образом:

[DllImport("lab8.dll")]

public static extern void memcopy32(byte[] dest, byte[] source, int block\_size);

[DllImport("lab8.dll")]

public static extern int sum\_int\_s(int a, int b);

[DllImport("lab8.dll")]

public static extern int sum\_int\_c(int a, int b);

[DllImport("lab8.dll")]

public static extern double mul\_double(double a, double b);

[DllImport("lab8.dll", EntryPoint = "@sum\_int\_f@16")]

public static extern int sum\_int\_f(int a, int b, int c, int d);

Динамическое подключение dll-библиотеки

Второй способ подключения dll-библиотеки является динамическим и осуществляется во время выполнения приложения. Сначала необходимо с помощью функции

HMODULE WINAPI LoadLibrary(char\* lpFileName)

загрузить в память библиотеку из файла с именем **lpFileName** и получить дескриптор dll-библиотеки. Затем, используя полученный дескриптор **hModule** и имя подпрограммы **lpProcName** получить с помощью функции

FARPROC WINAPI GetProcAddress (HMODULE hModule, char\* lpProcName)

адрес (указатель) нужной подпрограммы. Далее подпрограмму можно вызывать. Как только библиотека станет не нужна, её нужно выгрузить из памяти функцией **FreeLibrary**.

Пример динамического подключения dll-библиотеки:

// Тип – указатель на функцию

typedef void (\_stdcall \*func) (void\* a, void\* b, int count);

…

char dll\_name[] = "lab8.dll"; // Полный путь к библиотеке

HMODULE hModule = LoadLibraryA(dll\_name);

// Если dll-файл найден

if (hModule != NULL)

{

char func\_name[] = "memcopy32";

// Получить указатель на подпрограмму

func f = (func) GetProcAddress(hModule, func\_name);

if (f == NULL)

printf("Подпрограмма %s не найдена в библиотеке", func\_name);

else

{

int b[] = {1, 2, 3, 4, 5, 5, 4, 3, 2, 1}, a[10];

// Вызов подпрограммы

f(a, b, 10\*4);

}

// Освобождение библиотеки

FreeLibrary(hModule);

}

else

printf("dll-файл %s не найден", dll\_name);

Преимущество использования динамического подключения библиотеки перед статическим заключается в том, что если нужный файл с библиотекой не будет найден, можно принять какое-то решение о дальнейших действиях. Например, можно попытаться найти его или загрузить другой. В случае статического подключения, если dll-файл будет отсутствовать, программа просто не будет работать.

Ассемблерные вставки

Ещё одним простым способом встраивания ассемблерного кода в программы на C++ является использование ассемблерных вставок. Секция ассемблерного кода при этом просто записывается после ключевого слова **\_\_asm** в операторных скобках. Пример функции на C++:

void memcopy32\_2(void\* a, void\* b, int le1)

{

\_asm

{

mov ecx, le1

shr ecx, 2

cld

mov esi, b

mov edi, a

rep movsd

} }

**Задания для выполнения к работе**

1. Написать и отладить подпрограммы на masm32 в разных стилях вызова для решения задачи соответствующего варианта. Глобальные переменные в подпрограммах использовать не разрешается. Если нужна дополнительная память, выделять её в стеке.
2. Подпрограммы собрать и скомпилировать в виде dll-библиотеки. Библиотека должна содержать:
   * подпрограммы в стилях stdcall, cdecl, fastcall, написанные на ассемблере **без** явного перечисления аргументов в заголовке;
   * Подпрограммы в стилях stdcall, cdecl, написанные, наоборот, **с** перечислением аргументов в заголовке подпрограммы.
3. Подключить все подпрограммы из dll-библиотеки к проектам на C# и С++ статическим и динамическим способом. Убедиться в правильности вызова всех подпрограмм.
4. Написать подпрограмму для решения задачи варианта с использованием ассемблерной вставки на языке C++.
5. Написать подпрограммы для решения задачи варианта с использованием обычного высококровного языка C# и C++ (или любого другого).
6. Сравнить скорость выполнения полученных подпрограмм на одних и тех же тестовых данных. Для сравнения выбрать:

подпрограмму на ассемблере в masm32 (какую-нибудь одну из пяти), вызываемую из программы на языке C++ или C#; подпрограмму на C#; подпрограмму на C++; подпрограмму на С++ с использованием ассемблерной вставки. Построить на одной плоскости графики зависимости времени выполнения подпрограмм от длины массивов (не менее 10 точек для каждой подпрограммы). Для замера лучше передавать в подпрограммы массивы большой длины. Время замерять в миллисекундах с помощью API-функции GetTickCount(). Убедиться, что подпрограммы при одинаковых тестовых данных выдают одинаковый результат. Для заполнения массивов использовать генератор случайных чисел.

1. В отчёт включить исходный код и графики.
2. Сделать выводы по работе.

Варианты заданий

Если в заданиях базовые типы массивов *a* и *res* не совпадают (например, вариант №2), значит необходимо осуществить преобразование типа.

|  |  |
| --- | --- |
| Варианты 1-7  Сортировка части массива чисел с индексами от *start* до *end* включительно. Отсортированный массив (элементы *start*…*end*) записать в *res* и возвратить его адрес. Исходный массив *a* оставить без изменений. Под массив *res* зарезервировать память в необходимом размере, но не больше, чем нужно.  Пример: *a* = {4, 5, 4, 2, **5, 7, 5, 6, 5**, 3, 5, 6}, *start* = 4, *end* = 8;  *res* = {5, 5, 5, 6, 7} (сортировка по не убыванию). Длина массива *res* равна 5. | |
| 1 | Сортировка методом вставок по не убыванию.  float\* sort(float\* a, int start, int end, float\* res). |
| 2 | Пузырьковая сортировка по не возрастанию.  double\* sort(double\* res, float\* a, int start, int end). |
| 3 | Сортировка выбором по не убыванию.  int\* sort(int\* a, int start, int end, int\* res). |
| 4 | Сортировка методом вставок по не возрастанию.  short\* sort(short\* a, short\* res, int start, int end). |
| 5 | Пузырьковая сортировка по не убыванию.  int\* sort(int start, int end, int\* res, int\* a). |
| 6 | Сортировка выбором по не возрастанию.  float\* sort(float\* res, char start, int end, int\* a). |
| 7 | Сортировка выбором по не убыванию.  float\* sort(double\* a, int start, int end,  float\* res). |
| Варианты 8 - 16  Из массива *a* длиной *length* скопировать отрицательные числа в массив *neg\_res*, положительные – в массив *pos\_res*. Под массивы *neg\_res* и *pos\_res* в основной программе зарезервировать памяти столько, сколько занимает массив *a*. Полученные массивы отсортировать. Количество отрицательных чисел записать в выходной параметр *neg\_count*, положительных – в выходной параметр *pos\_count*. Исходный массив *a* оставить без изменений. Для удобства, можно реализовать в dll-библиотеке отдельную процедуру для сортировки одного массива.  Пример: *a* = {1, 3, 4, -5, 7, -2, -1, 3, 5, -5}, *length =* 10;  *pos\_res* = {1, 3, 3, 4, 5, 7} (сортировка по не убыванию);  *neg\_res* = {-1, -2, -5, -5} (сортировка по не возрастанию);  *pos\_count* = 6;  *neg\_count =* 4. | |
| 8 | Сортировка методом вставок по не убыванию.  int sort (int\* a, int length, int\* pos\_res,  int\* neg\_res, int\* neg\_count).  pos\_count возвратить. |
| 9 | Сортировка выбором по не возрастанию.  int sort (float\* a, int length, float\* pos\_res,  int\* pos\_count, float\* neg\_res).  neg\_count возвратить. |
| 10 | Пузырьковая сортировка по не возрастанию.  int sort (double\* a, int length, double\* pos\_res,  double\* neg\_res, int\* neg\_count).  pos\_count возвратить. |
| 11 | Сортировка методом вставок по не возрастанию.  int sort (float\* pos\_res, float\* neg\_res, int\* pos\_count, float\* a, int length).  **neg\_count** возвратить**.** |
| 12 | Сортировка выбором по не убыванию.  int sort (double\* pos\_res, double\* a, int length,  int\* pos\_count, double\* neg\_res).  neg\_count возвратить. |
| 13 | Пузырьковая сортировка по не убыванию.  int sort (int\* neg\_res, int\* a, int length, int\* pos\_res, int\* pos\_count).  neg\_count возвратить. |
| 14 | Сортировка методом вставок по не убыванию.  int sort (int\* pos\_count, double\* neg\_res, double\* a, int length, double\* pos\_res).  **neg\_count** возвратить. |
| 15 | Сортировка выбором по не убыванию.  int sort (int\* neg\_res, int\* pos\_res, int\* neg\_count, int\* a, int length).  pos\_count возвратить. |
| 16 | Пузырьковая сортировка по не убыванию.  int sort (float\* a, float\* neg\_res, float\* pos\_res, int length, int\* pos\_count).  neg\_count возвратить. |
| Варианты 17-25  Найти множество элементов массива *a* длиной *length* и его мощность *power* (количество элементов). Результат записать в *res*. Мощность возвратить и записать в выходной параметр *power*. Реализовать алгоритм таким образом, чтобы *res* был отсортирован. Массив *a* оставить без изменений.  Пример: *a* = {2, 5, 6, 4, 4, 9, 4, 5, 6, 1, 2, 5}, *length =* 12.  *res* = {1, 2, 4, 5, 6, 9}, *power* = 6. | |
| 17 | Сортировка по возрастанию.  int many(int\* a, int length, int\* res, int\* power). |
| 18 | Сортировка по убыванию.  int many(float\* res, float\* a, int length). |
| 19 | Сортировка по возрастанию.  int many(double\* res, int length, double\* a, int\* power). |
| 20 | Сортировка по убыванию. Оставить в *res* только чётные числа.  int many(short\* a, int length, short\* res,  int\* power). |
| 21 | Сортировка по убыванию.  int many(float\* a, int length, double\* res). |
| 22 | Сортировка по возрастанию.  int many(int length, float\* res, int\* a, int\* power). |
| 23 | Сортировка по убыванию. Оставить в *res* только числа, большие *e*1 и меньшие *e*2.  int many(short e1, int e2, int length, int\* res,  short\* a). |
| 24 | Сортировка по возрастанию. Оставить в *res* только числа, кратные *k*.  int many(int length, int\* res, short\* a, int k). |
| 25 | Сортировка по убыванию. Оставить в *res* только положительные числа.  int many(double\* res, int\* power, short\* a,  int length) |

Контрольные вопросы

1. Для чего используются dll-библиотеки?
2. Чем отличаются статический и динамический способы подключения dll-библиотеки?
3. Что такое декорирование названий подпрограмм?
4. Какие существуют стили вызова подпрограмм?
5. Что такое соглашения о вызовах (стили вызова)?
6. Какой стиль вызова используется компиляторами C++?
7. В чём особенность стиля вызова fastcall?
8. Что такое пролог и эпилог?
9. Для чего необходима инициализирующая функция dll-библиотеки?
10. Как передать в качестве аргумента подпрограмме число типа int\*; типа double?
11. Что такое ассемблерная вставка?
12. По каким правилам декорируются названия подпрограмм стилей вызова cdecl, stdcall, fastcall?
13. Какой стиль вызова используется в ООП?