Федеральное государственное бюджетное образовательное учреждение высшего

образования

«Чувашский государственный университет им. И.Н. Ульянова»

Кафедра вычислительной техники

Лабораторная работа № 4

СОЗДАНИЕ ДИНАМИЧЕСКИХ БИБЛИОТЕК

Вариант 20

Выполнил: Назаров Ярослав Сергеевич

студент группы ИВТ-42-23

Проверила: кандидат тех. наук

Андреева Антонина Аркадьевна

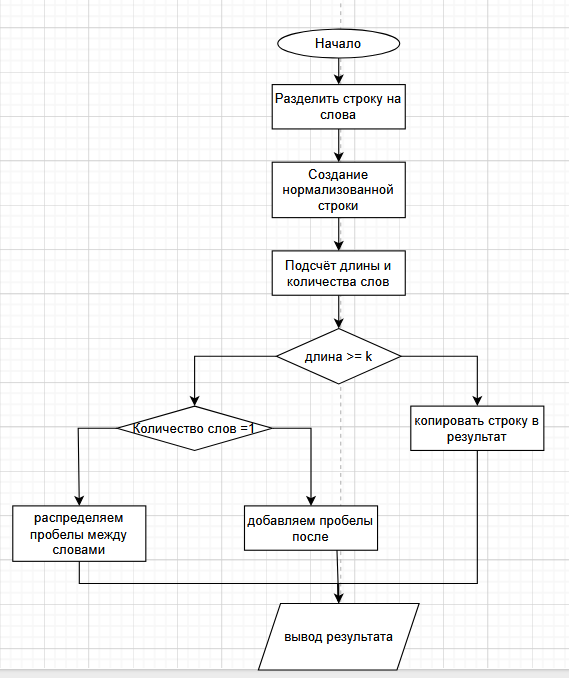
Чебоксары, 2025

**Цель работы:** научиться разрабатывать динамические библиотеки, использовать явное и неявное связывание с программой.

**Задание:** Разработать динамическую библиотеку, реализующую функ-  
ции в соответствии с заданным вариантом, и программу для де-  
монстрации ее возможностей. Использовать как явное, так и не-  
явное связывание. Примечание: pchar – строка ascii-символов,   
заканчивающаяся нулем.

**Выполнение работы:**

20. function InsBlanks(S: string; Len: byte): string. Возвращает   
строку длиной Len, в которой равномерно добавлены пробелы.   
Для этого строка циклически просматривается и после каждого   
слова в ней добавляется по одному пробелу до тех пор, пока не   
будет равенства длины строки и Len. Если длина S сразу больше   
или равна Len либо S – пустая строка, то ничего не изменяется.   
Примечание: пустой называется строка, не содержащая ни   
одного символа, байт длины которой равен нулю.



**Переход программы от Dos к Windows**

**1. Общие сведения**

Исходная программа была написана для среды **DOS**, с использованием 16-битного ассемблера TASM (режим .model small). При адаптации к **Windows** программа была переписана под 32-битную модель (.model flat, stdcall) и пересобрана средствами TASM32/MASM32.  
Основное назначение процедуры InsBlanks осталось неизменным — нормализация строки и распределение пробелов до требуемой длины.

**2. Изменения в организации программы**

1. **Модель памяти**
   * Было: .model small (разделение сегментов кода и данных, ограничение 64КБ).
   * Стало: .model flat, stdcall (единое адресное пространство 32-бит, соглашение вызова stdcall).
2. **Регистры и разрядность**
   * Было: использование 16-битных регистров (si, di, bx, cx, dx, bp).
   * Стало: использование 32-битных регистров (esi, edi, ebx, ecx, edx, ebp).
3. **Работа с указателями**
   * В DOS версии адреса слов сохранялись как **16-битные** значения (смещение), массив указателей формировался с шагом **2 байта**.
   * В Windows версии указатели стали **32-битными**, массив формируется с шагом **4 байта**.
4. **Стек и параметры**
   * В DOS (TASM16): параметры передавались через стек с базовым регистром bp. Схема: [bp+4] – первый параметр, [bp+6] – второй и т.д.
   * В Windows (TASM32): используется соглашение stdcall. Базовый регистр ebp применяется для доступа к аргументам: [ebp+8], [ebp+12], … Память под аргументы освобождается оператором ret 20 (5 аргументов × 4 байта).
5. **Сохранение регистров**
   * Было: pusha / popa (сохраняли 16-битные регистры).
   * Стало: pushad / popad (сохраняют 32-битные регистры).

**3. Изменения в логике и реализации**

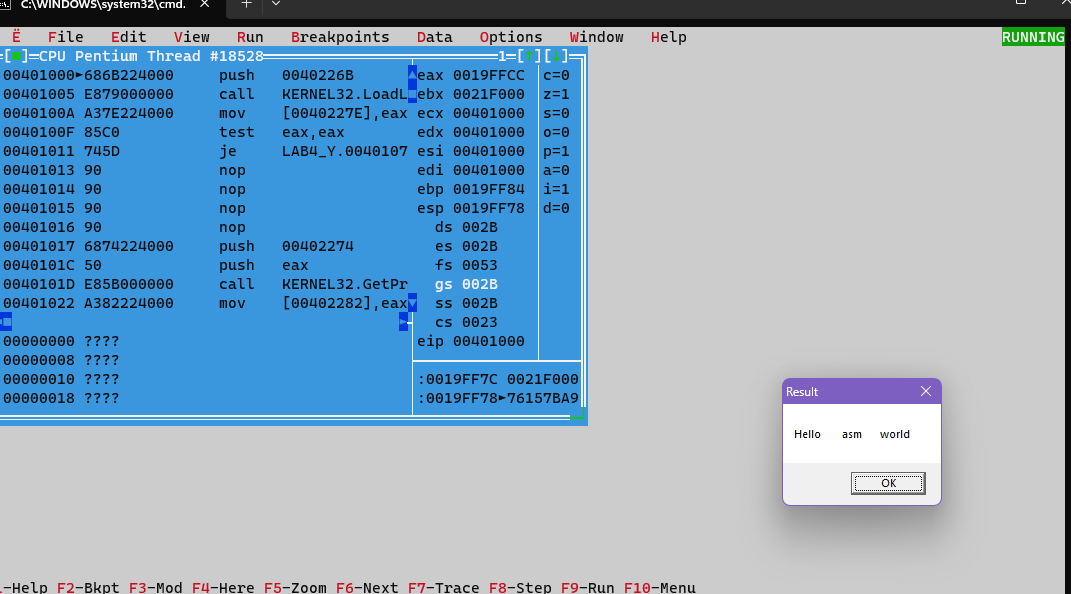
1. **Операции с массивами указателей**
   * DOS: при добавлении нового слова add di, 2.
   * Windows: при добавлении нового слова add edi, 4.
2. **Длина строки и пробелы**
   * DOS: операции выполнялись через 16-битные регистры (ax, cx, dx).
   * Windows: те же алгоритмы, но через 32-битные регистры (eax, ecx, edx).
3. **Копирование строк**
   * Алгоритм посимвольного копирования слов и вставки пробелов не изменился, но расширен на работу с 32-битными регистрами.
4. **Возврат из процедуры**
   * DOS: ret 10 (освобождение 5 аргументов по 2 байта).
   * Windows: ret 20 (освобождение 5 аргументов по 4 байта).

**4. Особенности перехода от DOS к Windows**

* **Адресация**: в DOS применялась сегментная модель, требовалось учитывать смещения в пределах сегмента. В Windows flat-модель обеспечивает прямую 32-битную адресацию.
* **Совместимость**: код DOS не может работать напрямую в Win32 без адаптации, так как 16-битные вызовы и модель памяти small не поддерживаются в защищённом режиме.
* **Соглашение вызова**: Windows требует явного указания соглашения (stdcall) для корректного взаимодействия с другими модулями и языками высокого уровня.
* **Регистр EBP**: используется как стандартная основа для доступа к аргументам процедуры.

**Пример выполнения программы на ассемблере:**  
Входная строка: Hello asm

Выходная строка:



**Вывод:** изучил подключение динамических библиотек, явное и неявное связывние и применил их на практике.

**prog.asm**

.386

.model flat, stdcall

public InsBlanks

.code

InsBlanks proc

    push ebp

    mov ebp, esp

    pushad

    ; Загрузка параметров:

    ; [ebp+8]  = input\_string

    ; [ebp+12] = k

    ; [ebp+16] = result\_buffer

    ; [ebp+20] = words\_buffer

    ; [ebp+24] = temp\_buffer

    ; 1. Разбиваем строку на слова

    mov esi, [ebp+8]   ; ESI = input\_string

    mov edi, [ebp+20]  ; EDI = words\_buffer

    xor ecx, ecx       ; ECX = счётчик слов

skip\_spaces:

    lodsb              ; Читаем символ

    cmp al, 0          ; Конец строки?

    je split\_end

    cmp al, ' '        ; Пропускаем пробелы

    je skip\_spaces

    ; Начало слова

    dec esi            ; Возвращаемся к первому символу

    mov [edi], esi     ; Сохраняем указатель

    add edi, 4         ; Следующий элемент массива (4 байта)

    inc ecx            ; Увеличиваем счётчик слов

read\_word:

    lodsb              ; Читаем символ

    cmp al, 0          ; Конец строки?

    je word\_end

    cmp al, ' '        ; Конец слова?

    jne read\_word

    mov byte ptr [esi-1], 0 ; Заменяем пробел на 0

    jmp skip\_spaces

word\_end:

    dec esi            ; Корректируем указатель

split\_end:

    push ecx           ; Сохраняем количество слов

    ; 2. Собираем нормализованную строку

    mov edi, [ebp+24]  ; EDI = temp\_buffer

    pop ecx            ; Восстанавливаем ECX (количество слов)

    jcxz empty\_temp    ; Если слов нет

    ; Копируем первое слово

    mov esi, [ebp+20]  ; ESI = words\_buffer

    mov esi, [esi]     ; ESI = адрес первого слова

copy\_first:

    mov al, [esi]

    test al, al        ; Конец слова?

    jz first\_done

    mov [edi], al      ; Копируем символ

    inc edi

    inc esi

    jmp copy\_first

first\_done:

    push ecx           ; Сохраняем количество слов

    pop ecx            ; Восстанавливаем ECX

    dec ecx            ; Оставшиеся слова

    jz temp\_done       ; Если только одно слово

    mov esi, [ebp+20]  ; ESI = words\_buffer

    add esi, 4         ; Переходим ко второму слову (4 байта)

next\_temp\_word:

    mov al, ' '        ; Добавляем пробел

    mov [edi], al

    inc edi

    mov ebx, [esi]     ; EBX = адрес слова

    add esi, 4         ; Следующее слово (4 байта)

copy\_word\_temp:

    mov al, [ebx]

    test al, al        ; Конец слова?

    jz word\_done\_temp

    mov [edi], al      ; Копируем символ

    inc edi

    inc ebx

    jmp copy\_word\_temp

word\_done\_temp:

    loop next\_temp\_word ; Повторяем для всех слов

temp\_done:

    mov byte ptr [edi], 0 ; Завершаем строку

    jmp after\_build

empty\_temp:

    mov byte ptr [edi], 0 ; Пустая строка

after\_build:

    ; 3. Вставляем пробелы для достижения длины k

    mov esi, [ebp+24]  ; ESI = temp\_buffer

    mov edi, [ebp+16]  ; EDI = result\_buffer

    mov ebx, [ebp+12]  ; EBX = k

    ; Проверка на пустую строку

    cmp byte ptr [esi], 0

    je copy\_direct

    ; Подсчёт длины и количества слов

    xor ecx, ecx       ; Длина строки

    xor edx, edx       ; Количество слов

    mov ah, 0          ; Флаг внутри слова

count\_loop:

    mov al, [esi]

    cmp al, 0          ; Конец строки?

    je end\_count

    inc ecx            ; Увеличиваем длину

    cmp al, ' '        ; Пробел?

    je space\_char

    ; Обработка буквы

    test ah, ah        ; Уже внутри слова?

    jnz not\_new\_word

    inc edx            ; Новое слово

    mov ah, 1          ; Устанавливаем флаг

not\_new\_word:

    jmp next\_char

space\_char:

    mov ah, 0          ; Сбрасываем флаг

next\_char:

    inc esi

    jmp count\_loop

end\_count:

    mov esi, [ebp+24]  ; Восстанавливаем начало строки

    cmp ecx, ebx       ; Текущая длина >= k?

    jae copy\_direct

    test edx, edx      ; Нет слов?

    jz copy\_direct

    cmp edx, 1         ; Одно слово?

    je handle\_single\_word

    ; Вычисляем пробелы для добавления

    mov eax, ebx       ; EAX = k

    sub eax, ecx       ; EAX = всего пробелов

    mov ecx, edx       ; ECX = количество слов

    dec ecx            ; Промежутки = слов - 1

    xor edx, edx

    div ecx            ; EAX = p, EDX = q

    ; Вставляем пробелы

    xor ecx, ecx       ; Счётчик промежутков

    mov ebx, eax       ; EBX = p

copy\_loop:

    mov al, [esi]      ; Читаем символ

    inc esi

    test al, al        ; Конец строки?

    jz end\_ins

    cmp al, ' '        ; Пробел?

    jne copy\_char

    ; Обработка пробела

    push ecx           ; Сохраняем счётчик

    mov ecx, 1         ; Базовый пробел

    add ecx, ebx       ; + p пробелов

    pop eax            ; EAX = текущий промежуток

    cmp eax, edx       ; Сравниваем с q

    jae no\_extra

    inc ecx            ; Добавляем дополнительный пробел

no\_extra:

    push eax           ; Сохраняем обратно

    mov al, ' '        ; Вставляем пробелы

insert\_spaces:

    mov [edi], al

    inc edi

    loop insert\_spaces

    pop ecx            ; Восстанавливаем счётчик

    inc ecx            ; Следующий промежуток

    jmp copy\_loop

copy\_char:

    mov [edi], al      ; Копируем символ

    inc edi

    jmp copy\_loop

end\_ins:

    mov byte ptr [edi], 0 ; Конец строки

    jmp done\_ins

handle\_single\_word:

    ; Обработка одного слова

    mov esi, [ebp+24]  ; Начало строки

    xor ecx, ecx       ; Длина слова

count\_single\_len:

    cmp byte ptr [esi], 0

    je end\_count\_single

    inc ecx

    inc esi

    jmp count\_single\_len

end\_count\_single:

    mov esi, [ebp+24]  ; Восстанавливаем начало

copy\_single\_word:

    mov al, [esi]

    test al, al        ; Конец слова?

    jz add\_tail\_spaces

    mov [edi], al

    inc edi

    inc esi

    jmp copy\_single\_word

add\_tail\_spaces:

    mov eax, [ebp+12]  ; EAX = k

    sub eax, ecx       ; Пробелы для добавления

    mov ecx, eax

    jle done\_tail      ; Если <=0, пропускаем

    mov al, ' '

    rep stosb          ; Заполняем пробелами

done\_tail:

    mov byte ptr [edi], 0 ; Конец строки

    jmp done\_ins

copy\_direct:

    ; Простое копирование

    mov al, [esi]

    mov [edi], al

    inc esi

    inc edi

    test al, al

    jnz copy\_direct

done\_ins:

    popad              ; Восстанавливаем регистры

    pop ebp

    ret 20             ; Возврат с очисткой стека (5 аргументов \* 4 байта)

InsBlanks endp

end

**prog.def**

LIBRARY prog

EXPORTS

InsBlanks

**prog.bat**

tasm32 /ml /l prog.asm

pause

tlink32 /Tpd /c prog.obj,,,import32.lib,prog.def

pause

implib prog.lib prog.dll

pause

**Неявное связывание**

**lab4\_n.bat**

tasm32 /ml /l lab4\_n.asm

pause

tlink32 /Tpe /aa /x /c lab4\_n.obj,,,import32.lib

pause

td32 lab4\_n.exe

**lab4\_n.asm**

.386

.model flat, stdcall

includelib import32.lib

includelib prog.lib       ; импорт-библиотека для prog.dll

extern MessageBoxA: near

extern ExitProcess: near

extern InsBlanks: near

.data

    input\_string  db 'Hello asm world',0

    k             dd 26

    result\_buffer db 256 dup(0)

    words\_buffer  dd 20 dup(0)

    temp\_buffer   db 256 dup(0)

    caption       db "Result",0

.code

main proc

    ; Вызов InsBlanks напрямую

    push offset temp\_buffer

    push offset words\_buffer

    push offset result\_buffer

    push k

    push offset input\_string

    call InsBlanks

    add esp, 20

    ; Вывод результата

    push 0

    push offset caption

    push offset result\_buffer

    push 0

    call MessageBoxA

    ; Завершение программы

    push 0

    call ExitProcess

main endp

end main

**Явное связывание**

**lab4\_y.asm**

.386

.model flat, stdcall

option casemap :none

includelib import32.lib

extern MessageBoxA: PROC

extern ExitProcess: PROC

extern LoadLibraryA: PROC

extern GetProcAddress: PROC

extern FreeLibrary: PROC

.data

    input\_string  db 'Hello asm world',0

    k             dd 26

    result\_buffer db 256 dup(0)

    words\_buffer  dd 20 dup(0)

    temp\_buffer   db 256 dup(0)

    caption       db "Result",0

    dllName       db "prog.dll",0

    funcName      db "InsBlanks",0

    hLib          dd 0

    pInsBlanks    dd 0

.code

main proc

    ; Загрузка DLL

    push offset dllName

    call LoadLibraryA

    mov hLib, eax

    test eax, eax

    jz exit\_program

    ; Получение адреса функции

    push offset funcName

    push eax

    call GetProcAddress

    mov pInsBlanks, eax

    test eax, eax

    jz unload\_dll

    ; Вызов InsBlanks через указатель

    push offset temp\_buffer

    push offset words\_buffer

    push offset result\_buffer

    push k

    push offset input\_string

    call pInsBlanks

    add esp, 20     ; чистим стек

    ; Вывод результата

    push 0

    push offset caption

    push offset result\_buffer

    push 0

    call MessageBoxA

unload\_dll:

    ; Освобождение библиотеки

    push hLib

    call FreeLibrary

exit\_program:

    push 0

    call ExitProcess

main endp

end main

**lab4\_y.bat**

tasm32 /ml /l lab4\_y.asm

pause

tlink32 /Tpe /aa /x /c lab4\_y.obj,,,import32.lib

pause

td32.exe lab4\_y.exe

1. Содержимое секции экспорта .dll файла
2. Содержимое секции импорта из exe файла при неявном связывании
3. Содержание .lib файла
4. Как происходит явное и неявное связывание
5. Сравнить явное и неявное связывание
6. Сравнить статическое связывание
7. Что такое связывание
8. Секция экспорта .edata будет содержать следующую ключевую информацию:

**1. Имя Библиотеки**

**Имя DLL: prog.DLL.**

Это имя используется исполняемыми файлами при неявном связывании (в секции импорта .idata) для идентификации библиотеки, из которой они хотят загрузить функции.

**2. Экспортируемые Функции**

**Имя функции: InsBlanks.**

Это имя является символом, который библиотека делает доступным для внешнего использования. Любая программа, которая связывается с prog.DLL, будет вызывать эту функцию.

**Структура Секции Экспорта (.edata)**

В PE-файле Windows (EXE/DLL) секция .edata структурирована следующим образом:

1. Каталог Экспорта (Export Directory Table): Главная структура, содержащая метаданные об экспортах:

* Имя DLL (prog.DLL).
* Базовый порядковый номер (Ordinal Base).
* Количество функций (в данном случае, как минимум 1, для InsBlanks).
* Указатели на другие таблицы:

1. Таблица Адресов Экспорта (Export Address Table - EAT): Массив адресов, содержащий RVA (Relative Virtual Address) каждой экспортируемой функции.

RVA — это смещение (адрес) функции (InsBlanks) относительно базового адреса, по которому prog.DLL загружается в память.

1. Таблица Имён Экспорта (Export Name Table - ENT): Список строковых имён всех экспортируемых функций, которые доступны по имени (например, InsBlanks).
2. Таблица Порядковых Номеров Имён (Export Name Ordinal Table - EOT): Каждая запись в этой таблице сопоставляет имя функции из ENT с порядковым номером (индексом) в EAT.
3. В общем случае, для каждой импортируемой DLL, секция .idata содержит:
   1. **Имя DLL** (например, KERNEL32.dll, USER32.dll).
   2. **Таблицу Дескрипторов Импорта (Import Directory Table - IDT)**. Одна запись для каждой DLL.
   3. **Таблицу Имён Импорта (Import Name Table - INT)**. Список всех функций, которые программа импортирует из данной DLL, либо по имени, либо по порядковому номеру (ordinal).
   4. **Таблицу Адресов Импорта (Import Address Table - IAT)**. Изначально (в файле) IAT совпадает с INT. **Загрузчик ОС** заполняет IAT фактическими адресами функций в памяти после того, как DLL будет загружена.

**Пример из lab4\_n.EXE**

На основе строк, обнаруженных в содержимом файла lab4\_n.EXE, можно сделать вывод о следующем содержимом секции импорта:

|  |  |  |
| --- | --- | --- |
| Элемент Секции Импорта | Данные из lab4\_n.EXE | Объяснение |
| **Имена DLL** | USER32.dll | Стандартная Windows DLL, содержащая функции для работы с пользовательским интерфейсом (окна, сообщения и т.д.). |
| KERNEL32.dll | Основная системная DLL, предоставляющая базовые функции ОС (управление процессами, памятью, файлами и т.д.). |
| prog.DLL | Пользовательская (или сторонняя) DLL, специфичная для данной программы. |
| **Импортируемые Функции** | MessageBoxA | Функция из USER32.dll для отображения диалогового окна. |
| ExitProcess | Функция из KERNEL32.dll для завершения текущего процесса. |
| InsBlanks | Вероятно, функция, импортируемая из prog.DLL или одной из других DLL, специфичная для работы программы. |

**3)** **Содержимое prog.lib (Библиотека Импорта)**

Файл prog.lib содержит заглушки (stub code) и метаданные, которые используются на этапе компоновки (link-time).

|  |  |  |
| --- | --- | --- |
| Элемент | Содержание в prog.lib | Роль |
| **Имя Импортируемой Функции** |  | InsBlanks 111 | Главный символ, который EXE-файл будет вызывать. Компоновщик использует это имя для создания записи в таблице импорта. |
| **Имя Исходной DLL** |  | prog.DLL 2 | Указывает компоновщику, из какой именно динамической библиотеки необходимо импортировать функцию InsBlanks. Эта строка будет помещена в секцию .idata EXE-файла. |
| **Заглушка (Stub Code)** |  | *Невидимый бинарный код* 3 | Содержит минимальный код (например, инструкцию JMP), который компоновщик встраивает в EXE-файл. При запуске программы этот код будет указывать на запись в **Таблице Адресов Импорта (IAT)**. |

**Роль Библиотеки Импорта**

Основная задача prog.lib — связать вызовы функций в вашем EXE-файле с местом в prog.DLL, где эти функции реально находятся, но **без включения** фактического кода функций:

1. **Для Компоновщика (Linker):** Компоновщик читает prog.lib, чтобы узнать:
   * Какие функции (InsBlanks) требуются.
   * Из какой DLL (prog.DLL) их нужно взять.
   * Эту информацию компоновщик записывает в секцию .idata EXE-файла.
2. **Для Загрузчика ОС (OS Loader):** Когда EXE-файл запускается, загрузчик использует информацию из .idata (которую создал компоновщик) для:
   * Поиска и загрузки prog.DLL.
   * Поиска фактического адреса функции InsBlanks в памяти внутри prog.DLL.
   * Обновления **IAT** EXE-файла этим фактическим адресом.

**4) Как происходит явное и неявное связывание**

**Неявное связывание**

* Указывается на этапе компиляции и линковки программы.
* В коде объявляются внешние процедуры (extrn), а при линковке указываются .lib-файлы.
* Линковщик формирует в исполняемом файле таблицу импортов.
* При запуске программы операционная система автоматически загружает необходимые динамические библиотеки (DLL) и связывает адреса функций.

**Явное связывание:**

* Выполняется во время работы программы самим приложением.
* Программа вызывает системные функции загрузки библиотеки (LoadLibrary ), получает её дескриптор.
* Адрес нужной функции извлекается через GetProcAddress .
* Вызов выполняется через указатель на функцию.

**5) Сравнить явное и неявное связывание**

|  |  |  |
| --- | --- | --- |
| **Характеристика** | **Неявное связывание** | **Явное связывание** |
| Когда выполняется | При запуске программы | Во время работы программы |
| Кто загружает библиотеку | Операционная система | Программа самостоятельно |
| Ошибка при отсутствии DLL | Программа не запустится | Можно обработать и продолжить работу |
| Простота реализации | Проще в коде | Сложнее, требует дополнительного кода |
| Гибкость | Жёсткая привязка к конкретной библиотеке | Можно загружать разные версии, плагины |
| Производительность | Быстрее вызов функций | Незначительное замедление при первом вызове |

**6) Сравнить статическое связывание**

**Статическое связывание**:

* Библиотека встраивается в исполняемый файл на этапе линковки.
* В результате формируется единый exe без внешних зависимостей.
* Плюсы: программа не зависит от внешних файлов, проще распространять.
* Минусы: увеличивается размер исполняемого файла, обновление библиотеки требует пересборки всего приложения.

**Сравнение с динамическим связыванием (неявным/явным):**

* Динамическое связывание уменьшает размер exe и позволяет обновлять библиотеку отдельно от программы.
* Статическое связывание даёт независимость от окружения, но лишает гибкости.

**7) Что такое связывание**

**Связывание (linking)** — это процесс объединения объектных модулей программы и внешних библиотек в единый исполняемый файл.  
Задача связывания — разрешить внешние ссылки, то есть подставить реальные адреса для всех функций и данных, объявленных как внешние (extern).

Существует три основных вида связывания:

1. **Статическое** — код библиотеки встраивается в exe.
2. **Динамическое неявное** — библиотеки подключаются автоматически при запуске.
3. **Динамическое явное** — библиотеки подгружаются вручную в ходе работы программы.