Міністерство освіти і науки України

Західноукраїнський національний університет

Факультет комп’ютерних інформаційних технологій

Кафедра Комп'ютерної інженерії

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

до виконання лабораторних робіт
з дисципліни “Системне програмування”

для студентів спеціальності
123 “Комп’ютерна інженерія ”

2022

Методичні рекомендації до виконання лабораторних робіт з дисципліни “Системне програмування” для студентів спеціальності 123 "Комп’ютерні системи та мережі" / Ю.М. Батько / Тернопіль: ЗУНУ, 2022.– 70 с.

Укладач: Ю.М. Батько, к.т.н., доцент.

Відповідальний за випуск: Березький О.М., д.т.н., професор.

Рецензенти:
Михалик Д.М. - к.т.н., доцент кафедри програмної інженерії Тернопільського національного технічного університету ім. Івана Полюя

Івасєв С.В. – к.т.н., доцент кафедри КБ ЗУНУ

Методичні рекомендації розглянуто та рекомендовано до друку на засіданні кафедри комп’ютерної інженерії протокол № 7 від 22 лютого 2022 р.

ЗМІСТ

ВСТУП

Від моменту появи 1995 року, від початку глобальної комп'ютеризації і до сьогодні – мова програмування Java стабільно користується попитом на ринку. З того часу Java довела свою затребуваність.

Java – це універсальна мова програмуваня. Сьогодні з її допомогою можна створювати програмне забезпечення як для комп'ютерів, так і для мобільних пристроїв. Віртуальна Java (JVM) популярна й для інших мов і платформ. Більше 20 років розвитку на чолі з геніальними корпораціями привели до створення налагодженої моделі. Так що зараз існують такі мови програмування, які Scala, Groovy і Jruby, котрі компілюються з байт-кодом JVM. І знання Java допоможе Вам вивчити ці мови, оскільки в них будуть часто використовуватися інтерфейси програмування додатків Java. Отже, щоб відбувся повний відхід від даної платформи, потрібен далеко не один рік. За цей час ви можете вивчити цю мову програмування, створити немало продуктів і заробити на своїх знаннях.

Продукти Java можна зустріти буквально скрізь. Завдяки простоті та надійності Java використовують у різних сферах життя: для розробки ПЗ в державній сфері, в науці, освіті, сфері охорони здоров'я, в приватному секторі при створенні програм для трейдингу, серверні додатки для банкінгу та для багатьох інших корпоративних і ентерпрайз цілей. Розробка оновлень постійно триває. Актуальним залишається і старий код, тому немає потреби пристосовуватися до чогось кардинально нового.

Лабораторна робота 1

Тема: Створення віконного додатку засобами функцій Win API.

Мета: Ознайомитися з структурою та принципами функціонування віконних програмних додатків в ОС Windows створених за допомогою функцій Win API.

1. Теоретичні відомості

API (Application Programming Interface -інтерфейс прикладних програм) - це безліч функцій, організованих, звичайно, у вигляді DLL. Функції API дозволяють організувати інтерфейс між прикладною програмою і середовищем, в якому працює ця програма. Виклик функцій API дозволяє програмі отримуватидоступ до ресурсів середовища і управляти її роботою. Як правило,API задає стандарт взаємодії середовища і прикладної програми.

Win32 - це назва інтерфейсу, орієнтованого на 32-х розрядні додатки і реалізованого на таких відомих платформах як Windows 95 – Windows10. Функції, що становлять цей інтерфейс, дозволяють прикладній програмі отримувати доступ до ресурсів операційної системи і керувати її роботою. Більш ранні версії Windows використовують інтерфейс, відомий якWin16. Звичайно, не всі функції, складові інтерфейс Win32, реалізовані в повній мірі на всіх платформах, так що виклик одній і тій же функції під NT призведе до певного результату, а під Windows 10 працює як виклик заглушки . Будь-який з додатків, що працює в середовищі Windows , прямо або побічно викликає функції, що входять в Win32API.

Функції, складові Win32 інтерфейс , організовані у вигляді декількох динамічно підключаються бібліотек (DLL) і виконуваних файлів. Говорячи про Win32API , слід в першу чергу згадати три основні бібліотеки:

Kernel32.dll. Ця бібліотека призначена для роботи з об'єктами ядра операційної системи і її функції дозволяють управляти пам'яттю і іншими системними ресурсами.

User32.dll. Тут зосереджені функції для управління вікнами - основним видом об'єктів операційної системи. Обробка повідомлень, робота з меню, таймерами, все це виконують функції цієї DLL .

GDI32.dll. Ця бібліотека, що забезпечує графічний інтерфейс операційної системи (Graphics Device Interface). Функції управління виводу на екран дисплея, управління виводу принтера, функції для роботи зі шрифтами - всі вони входять до складу цієї бібліотеки.

Зауважте, WinAPI функції знаходяться не тільки в цих бібліотеках. З іншого боку API функції не обов'язково входять до складу Win32 інтерфейсу. наприклад, MAPI інтерфейс (Messaging Application Programming Interface) складають функції, призначені для обробки повідомлень електронної пошти,

TAPI (Telephone API) - функції роботи з телефонними повідомленнями. MAPI ,TAPI , також як і Win32 це деякий набір функцій, що задає певний стандарт взаємодії

Функції, що утворюють API, зазвичай, організовані у вигляді DLL - динамічно підключаються бібліотеках. Одна з переваг DLL полягає в тому, що, скільки б додатків (процесів) не працював з функціями однієї і тієї ж DLL , код DLL існує в єдиному екземплярі.

У своїй роботі ОС Windows використовує ряд типів даних, що побудовані на стандартних типах даних мови C++. Нижче наведено список типів даних для їх більшого розуміння.

Термін	Опис
ATOM	Атом Цей тип оголошений в **Windef.h** як показано нижче: typedef WORD ATOM;
BOOL	Булева змінна (повинна бути ІСТИНА (TRUE) або БРЕХНЯ (FALSE)). Цей тип оголошений в **Windef.h** як показано нижче: typedef int BOOL;
BOOLEAN	Булева змінна (повинна бути ІСТИНА (TRUE) або БРЕХНЯ (FALSE)). Цей тип оголошений в **Winnt.h** як показано нижче: typedef BYTE BOOLEAN;
BYTE	Байт (8 біт). Цей тип оголошений в **Windef.h** як показано нижче: typedef unsigned char BYTE;
CALLBACK	Угода про виклики для функцій повторного виклику. Цей тип оголошений в **Windef.h** як показано нижче: #define CALLBACK __stdcall
CHAR	8-бітовий символ Windows (**ANSI**). Цей тип оголошений в **Winnt.h** як показано нижче: typedef char CHAR;
COLORREF	Червоний (**r** ed), зелений (**g** reen), блакитний (**b** lue) (**RGB**) значення кольору (32 біта). Цей тип оголошений в **Windef.h** як показано нижче: typedef DWORD COLORREF;
CONST	Змінна, значення якої залишається постійним в ході виконання програми. Цей тип оголошений в **Windef.h** як показано нижче: #define CONST const
DWORD	32-розрядний беззнаковое ціле число. Цей тип оголошений в **Windef.h** як показано нижче: typedef unsigned long DWORD;
DWORDLONG	64-розрядний беззнаковое ціле число. Цей тип оголошений в **Winnt.h** як показано нижче: typedef ULONGLONG DWORDLONG;
DWORD_PTR	Тип беззнаковий дальній для точності покажчика. Використовується тоді, коли проводиться приведення покажчика до дальнього типу, щоб виконати арифметичні операції над покажчиками. (Також зазвичай використовується для загальних 32-розрядних параметрів, які були розширені до 64 бітів в 64-розрядному Windows). Цей тип оголошений в **Basetsd.h** як показано нижче: typedef ULONG_PTR DWORD_PTR;
DWORD32	32-розрядний беззнаковое ціле число. Цей тип оголошений в **Basetsd.h** як показано нижче: typedef unsigned int DWORD32;
DWORD64	64-розрядний беззнаковое ціле число. Цей тип оголошений в **Basetsd.h** як показано нижче: typedef unsigned __int64 DWORD64;
FLOAT	Змінна з плаваючою точкою. Цей тип оголошений в **Windef.h** як показано нижче: typedef float FLOAT;
HACCEL	Дескриптор **таблиці прискорювачів** (accelerator table). Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HACCEL;

HANDLE	Дескриптор об'єкта. Цей тип оголошений в **Winnt.h** як показано нижче: typedef PVOID HANDLE;
HBITMAP	Дескриптор **точкового малюнка** (bitmap) . Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HBITMAP;
HBRUSH	Дескриптор **пензля** . Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HBRUSH;
HCOLORSPACE	Дескриптор **колірного простору** (color space) .Цей тип оголошений в **Windef.h** як показано нижче: if (WINVER> = 0x0400) typedef HANDLE HCOLORSPACE;
HCONV	Дескриптор динамічного обміну даними (**DDE**) в режимі діалогу. Цей тип оголошений в **Ddeml.h** як показано нижче: typedef HANDLE HCONV;
HCONVLIST	Дескриптор списку **DDE** в режимі діалогу. Цей тип оголошений в **Ddeml.h** як показано нижче: typedef HANDLE HCONVLIST;
HCURSOR	Дескриптор **курсора** . Цей тип оголошений в **Windef.h** як показано нижче: typedef HICON HCURSOR;
HDC	Дескриптор **контексту пристрою** (**DC**). Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HDC;
HDDEDATA	Дескриптор даних **DDE** . Цей тип оголошений в **Ddeml.h** як показано нижче: typedef HANDLE HDDEDATA;
HDESK	Дескриптор робочого столу . Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HDESK;
HDROP	Дескриптор структури вставки всередину. Цей тип оголошений в **Shellapi.h** як показано нижче: typedef HANDLE HDROP;
HDWP	Дескриптор структури відкладеної позиції вікна. Цей тип оголошений в **Winuser.h** як показано нижче: typedef HANDLE HDWP;
HENHMETAFILE	Дескриптор вдосконаленого метафайлу (enhanced metafile). Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HENHMETAFILE;
HFILE	Дескриптор відкритого файлу за допомогою **OpenFile** , а не **CreateFile** . Цей тип оголошений в **Windef.h** як показано нижче: typedef int HFILE;
HFONT	Дескриптор **шрифту** . Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HFONT;
HGDIOBJ	Дескриптор об'єкта **GDI** . Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HGDIOBJ;
HGLOBAL	Дескриптор блоку глобальної пам'яті. Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HGLOBAL;
HHOOK	Дескриптор **hook-точки** . Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HHOOK;
HICON	Дескриптор піктограми . Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HICON;
HINSTANCE	Дескриптор примірника виконуваного модуля. Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HINSTANCE;
HKEY	Дескриптор ключа реєстру. Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HKEY;
HKL	Ідентифікатор введення даних національної мови. Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HKL;
HLOCAL	Дескриптор блоку локальної пам'яті. Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HLOCAL;

HMENU	Дескриптор **меню** . Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HMENU;
HMETAFILE	Дескриптор метафайлу . Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HMETAFILE;
HMODULE	Дескриптор модуля. Це значення - базовий адреса модуля Цей тип оголошений в **Windef.h** як показано нижче: typedef HINSTANCE HMODULE;
HMONITOR	Дескриптор монітора. Цей тип оголошений в **Windef.h** як показано нижче: if (WINVER> = 0x0500) typedef HANDLE HMONITOR;
HPALETTE	Дескриптор палітри . Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HPALETTE;
HPEN	Дескриптор **пера** . Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HPEN;
HRESULT	Повертає код використовуваний інтерфейсом. Він дорівнює нулю після успішного завершення, а не нуль позначає код помилки або інформацію про стан. Цей тип оголошений в **Winnt.h** як показано нижче: typedef LONG HRESULT;
HRGN	Дескриптор регіону . Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HRGN;
HRSRC	Дескриптор ресурсу. Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HRSRC;
HSZ	Дескриптор рядки **DDE** . Цей тип оголошений в **Ddeml.h** як показано нижче: typedef HANDLE HSZ;
HWINSTA	Дескриптор віконної станції . Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE WINSTA;
HWND	Дескриптор **вікна** . Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE HWND;
INT	32-розрядний знакове ціле число. Цей тип оголошений в **Windef.h** як показано нижче: typedef int INT;
INT_PTR	Цілий знаковий тип для точності покажчика. Використовується тоді, коли проводиться приведення покажчика до цілого типу, щоб виконати арифметичні операції над покажчиком. Цей тип оголошений в **Basetsd.h** як показано нижче: #if defined (_WIN64) typedef __int64 INT_PTR; #else typedef int INT_PTR;
INT32	32-розрядний знакове ціле число. Цей тип оголошений в **Basetsd.h** як показано нижче: typedef signed int INT32;
INT64	64-розрядний знакове ціле число. Цей тип оголошений в **Basetsd.h** як показано нижче: typedef signed __int64 INT64;
LANGID	Ідентифікатор мови. Цей тип оголошений в **Winnt.h** як показано нижче: typedef WORD LANGID;
LCID	Ідентифікатор національної мови Цей тип оголошений в **Winnt.h** як показано нижче: typedef DWORD LCID;
LCTYPE	Тип інформації про національну мову. Цей тип оголошений в **Winnls.h** як показано нижче: typedef DWORD LCTYPE;
LGRPID	Ідентифікатор групи мов. Список дивись в описі **EnumLanguageGroupLocales** . Цей тип оголошений в **Winnls.h** як показано нижче: typedef DWORD LGRPID;
LONG	32-розрядний знакове ціле число. Цей тип оголошений в **Winnt.h** як показано нижче: typedef long LONG;
LONGLONG	64-розрядний знакове ціле число. Цей тип оголошений в **Winnt.h** як показано нижче: typedef __int64 LONGLONG; #else typedef double LONGLONG;

LONG_PTR	Далекий знаковий тип для точності покажчика. Використовується тоді, коли проводиться приведення покажчика до дальнього типу, щоб виконати арифметичні операції над покажчиками. Цей тип оголошений в **Basetsd.h** як показано нижче: typedef long LONG_PTR;
LONG32	32-розрядний знакове ціле число. Цей тип оголошений в **Basetsd.h** як показано нижче: typedef signed int LONG32;
LONG64	64-розрядний знакове ціле число. Цей тип оголошений в **Basetsd.h** як показано нижче: typedef __int64 LONG64;
LPARAM	Параметр повідомлення. Цей тип оголошений в **Windef.h** як показано нижче: typedef LONG_PTR LPARAM;
LPBOOL	Покажчик на **BOOL** . Цей тип оголошений в **Windef.h** як показано нижче: typedef BOOL * LPBOOL;
LPBYTE	Покажчик на **BYTE** . Цей тип оголошений в **Windef.h** як показано нижче: typedef BYTE * LPBYTE;
LPCOLORREF	Покажчик на значення **COLORREF** . Цей тип оголошений в **Windef.h** як показано нижче: typedef DWORD * LPCOLORREF;
LPCSTR	Покажчик на строкову константу з нулем в кінці 8-розрядних символів (**ANSI**) Windows. Цей тип оголошений в **Winnt.h** як показано нижче: typedef CONST CHAR * LPCSTR;
LPCTSTR	Якщо визначено **Unicode** , то тип **LPCWSTR,** інакше **LPCTSTR** . Цей тип оголошений в **Winnt.h** як показано нижче: #ifdef UNICODE typedef LPCWSTR LPCTSTR; #else typedef LPCSTR LPCTSTR;
LPCVOID	Покажчик на константу будь-якого типу. Цей тип оголошений в **Windef.h** як показано нижче: typedef CONST void * LPCVOID;
LPCWSTR	Покажчик на строкову константу з нулем в кінці з 16-бітових символів **Unicode** . Цей тип оголошений в **Winnt.h** як показано нижче: typedef CONST WCHAR * LPCWSTR;
LPDWORD	Покажчик на **DWORD** . Цей тип оголошений в **Windef.h** як показано нижче: typedef DWORD * LPDWORD;
LPHANDLE	Покажчик на **HANDLE (ДЕСКРИПТОР)** . Цей тип оголошений в **Windef.h** як показано нижче: typedef HANDLE * LPHANDLE;
LPINT	Покажчик на **INT** . Цей тип оголошений в **Windef.h** як показано нижче: typedef int * LPINT;
LPLONG	Покажчик на **LONG** . Цей тип оголошений в **Windef.h** як показано нижче: typedef long * LPLONG;
LPSTR	Покажчик на рядок з нулем в кінці з 8-бітових символів Windows (**ANSI**). Цей тип оголошений в **Winnt.h** як показано нижче: typedef CHAR * LPSTR;
LPTSTR	Якщо визначено як **Unicode** , то тип **LPWSTR,** інакше **LPSTR** . Цей тип оголошений в **Winnt.h** як показано нижче: #ifdef UNICODE typedef LPWSTR LPTSTR; #else typedef LPSTR LPTSTR;
LPVOID	Покажчик на будь-який тип. Цей тип оголошений в **Windef.h** як показано нижче: typedef void * LPVOID;
LPWORD	Покажчик на **WORD** . Цей тип оголошений в **Windef.h** як показано нижче: typedef WORD * LPWORD;
LPWSTR	Покажчик на рядок з нулем в кінці з 16-бітових символів **Unicode** . Цей тип оголошений в **Winnt.h** як показано нижче: typedef WCHAR * LPWSTR;
LRESULT	Знаковий результат обробки повідомлення. Цей тип оголошений в **Windef.h** як показано нижче:

	typedef LONG_PTR LRESULT;
PBOOL	Покажчик на **BOOL** . Цей тип оголошений в **Windef.h** як показано нижче: typedef BOOL * PBOOL;
PBOOLEAN	Покажчик на **BOOL** . Цей тип оголошений в **Winnt.h** як показано нижче: typedef BOOLEAN * PBOOLEAN;
PBYTE	Покажчик на **BYTE** . Цей тип оголошений в **Windef.h** як показано нижче: typedef BYTE * PBYTE;
PCHAR	Покажчик на **CHAR** . Цей тип оголошений в **Winnt.h** як показано нижче: typedef CHAR * PCHAR;
PCSTR	Покажчик на строкову константу з нулем в кінці з 8-розрядних символів (**ANSI**) **Windows** . Цей тип оголошений в **Winnt.h** як показано нижче: typedef CONST CHAR * PCSTR;
PCTSTR	Якщо визначено в **Unicode** , то тип **PCWSTR,** інакше **PCSTR** . Цей тип оголошений в **Winnt.h** як показано нижче: #ifdef UNICODE typedef LPCWSTR PCTSTR; #else typedef LPCSTR PCTSTR;
PCWSTR	Покажчик на строкову константу з нулем в кінці з 16-бітових символів **Unicode** . Цей тип оголошений в **Winnt.h** як показано нижче: typedef CONST WCHAR * PCWSTR;
PDWORD	Покажчик на **DWORD** . Цей тип оголошений в **Windef.h** як показано нижче: typedef DWORD * PDWORD;
PDWORDLONG	Покажчик на **DWORDLONG** . Цей тип оголошений в **Winnt.h** як показано нижче: typedef DWORDLONG * PDWORDLONG;
PDWORD_PTR	Покажчик на **DWORD_PTR** . Цей тип оголошений в **Basetsd.h** як показано нижче: typedef DWORD_PTR * PDWORD_PTR;
PDWORD32	Покажчик на **DWORD32** . Цей тип оголошений в **Basetsd.h** як показано нижче: typedef DWORD32 * PDWORD32;
PDWORD64	Покажчик на **DWORD64** . Цей тип оголошений в **Basetsd.h** як показано нижче: typedef DWORD64 * PDWORD64;
PFLOAT	Покажчик на **FLOAT** . Цей тип оголошений в **Windef.h** як показано нижче: typedef FLOAT * PFLOAT;
PHANDLE	Покажчик на **HANDLE** (**ДЕСКРИПТОР**). Цей тип оголошений в **Winnt.h** як показано нижче: typedef HANDLE * PHANDLE;
PHKEY	Покажчик на **HKEY** . Цей тип оголошений в **Windef.h** як показано нижче: typedef HKEY * PHKEY;
PINT	Покажчик на **INT** . Цей тип оголошений в **Windef.h** як показано нижче: typedef int * PINT;
PINT_PTR	Покажчик на **INT_PTR.** Цей тип оголошений в **Basetsd.h** як показано нижче: typedef INT_PTR * PINT_PTR;
PINT32	Покажчик на **INT32.** Цей тип оголошений в **Basetsd.h** як показано нижче: typedef INT32 * PINT32;
PINT64	Покажчик на **INT64.** Цей тип оголошений в **Basetsd.h** як показано нижче: typedef INT64 * PINT64;
PLCID	Покажчик на **LCID** . Цей тип оголошений в **Winnt.h** як показано нижче: typedef PDWORD PLCID;
PLONG	Покажчик на **LONG** . Цей тип оголошений в **Winnt.h** як показано нижче: typedef LONG * PLONG;
PLONGLONG	Покажчик на **LONGLONG .** Цей тип оголошений в **Winnt.h** як показано нижче: typedef LONGLONG * PLONGLONG;
PLONG_PTR	Покажчик на **LONG_PTR** . Цей тип оголошений в **Basetsd.h** як показано нижче: typedef LONG_PTR * PLONG_PTR;

PLONG32	Покажчик на **LONG32** . Цей тип оголошений в **Basetsd.h** як показано нижче: typedef LONG32 * PLONG32;
PLONG64	Покажчик на **LONG64** . Цей тип оголошений в **Basetsd.h** як показано нижче: typedef LONG64 * PLONG64;
POINTER_32	32-розрядний покажчик. На 32-розрядної системі, це - рідний покажчик. На 64-розрядної системі, це -усеченний 64-розрядний покажчик. Цей тип оголошений в **Basetsd.h** як показано нижче: #if defined (_WIN64) #define POINTER_32 __ptr32 #else #define POINTER32
POINTER_64	64-розрядний покажчик. На 64-розрядної системі, це - рідний покажчик. На 32-розрядної системі, це - знаковий розширений 32-розрядний покажчик. Цей тип оголошений в **Basetsd.h** як показано нижче: #define POINTER_64 __ptr64
PSHORT	Покажчик на **SHORT** . Цей тип оголошений в **Winnt.h** як показано нижче: typedef SHORT * PSHORT;
PSIZE_T	Покажчик на **SIZE_T.** Цей тип оголошений в **Basetsd.h** як показано нижче: typedef SIZE_T * PSIZE_T;
PSSIZE_T	Покажчик на **SSIZE_T.** Цей тип оголошений в **Basetsd.h** як показано нижче: typedef SSIZE_T * PSSIZE_T;
PSTR	Покажчик на строкову константу з нулем в кінці 8-розрядних символів (**ANSI**) **Windows** .Цей тип оголошений в **Winnt.h** як показано нижче: typedef CHAR * PSTR;
PTBYTE	Покажчик на **TBYTE** . Цей тип оголошений в **Winnt.h** як показано нижче: typedef TBYTE * PTBYTE;
PTCHAR	Покажчик на **TCHAR** . Цей тип оголошений в **Winnt.h** як показано нижче: typedef TCHAR * PTCHAR;
PTSTR	Якщо визначено як **Unicode** , то тип **PWSTR,** інакше **PSTR** . Цей тип оголошений в **Winnt.h** як показано нижче: #ifdef UNICODE typedef LPWSTR PTSTR; #else typedef LPSTR PTSTR;
PUCHAR	Покажчик на **UCHAR** . Цей тип оголошений в **Windef.h** як показано нижче: typedef UCHAR * PUCHAR;
PUINT	Покажчик на **UINT** . Цей тип оголошений в **Windef.h** як показано нижче: typedef UINT * PUINT;
PUINT_PTR	Покажчик на **UINT_PTR.** Цей тип оголошений в **Basetsd.h** як показано нижче: typedef UINT_PTR * PUINT_PTR;
PUINT32	Покажчик на **UINT32.** Цей тип оголошений в **Basetsd.h** як показано нижче: typedef UINT32 * PUINT32;
PUINT64	Покажчик на **UINT64.** Цей тип оголошений в **Basetsd.h** як показано нижче: typedef UINT64 * PUINT64;
PULONG	Покажчик на **ULONG** . Цей тип оголошений в **Windef.h** як показано нижче: typedef ULONG * PULONG;
PULONGLONG	Покажчик на **ULONGLONG.** Цей тип оголошений в **Windef.h** як показано нижче: typedef ULONGLONG * PULONGLONG;
PULONG_PTR	Покажчик на **ULONG_PTR.** Цей тип оголошений в **Basetsd.h** як показано нижче: typedef ULONG_PTR * PULONG_PTR;
PULONG32	Покажчик на **ULONG32.** Цей тип оголошений в **Basetsd.h** як показано нижче: typedef ULONG32 * PULONG32;
PULONG64	Покажчик на **ULONG64.** Цей тип оголошений в **Basetsd.h** як показано нижче: typedef ULONG64 * PULONG64;
PUSHORT	Покажчик на **USHORT** . Цей тип оголошений в **Windef.h** як показано нижче: typedef USHORT * PUSHORT;

PVOID	Покажчик на будь-який тип. Цей тип оголошений в **Winnt.h** як показано нижче: typedef void * PVOID;
PWCHAR	Покажчик на **WCHAR** . Цей тип оголошений в **Winnt.h** як показано нижче: typedef WCHAR * PWCHAR;
PWORD	Покажчик на **WORD** . Цей тип оголошений в **Windef.h** як показано нижче: typedef WORD * PWORD;
PWSTR	Покажчик на рядок з нулем в кінці з 16-бітових символів **Unicode** . Цей тип оголошений в **Winnt.h** як показано нижче: typedef WCHAR * PWSTR;
SC_HANDLE	Дескриптор менеджера сервісного управління базою даних. Цей тип оголошений в **Winsvc.h** як показано нижче: typedef HANDLE SC_HANDLE;
SC_LOCK	Дескриптор менеджера сервісного управління блокуванням бази даних. Цей тип оголошений в **Winsvc.h** як показано нижче: typedef LPVOID SC_LOCK;
SERVICE_STATUS_ HANDLE	Значення дескриптора стану модуля обслуговування. Цей тип оголошений в **Winsvc.h** як показано нижче: typedef HANDLE SERVICE_STATUS_HANDLE;
SHORT	Коротке ціле число (16 біт). Цей тип оголошений в **Winnt.h** як показано нижче: typedef short SHORT;
SIZE_T	Максимальне число байтів, на які покажчик може вказувати. Використовується для рахунку, який повинен охопити повністю діапазон покажчика. Цей тип оголошений в **Basetsd.h** як показано нижче: typedef ULONG_PTR SIZE_T;
SSIZE_ T	Знаковий **SIZE_T** . Цей тип оголошений в **Basetsd.h** як показано нижче: typedef LONG_PTR SSIZE_T
TBYTE	Якщо визначено як **U nicode** , то тип **WCHAR,** інакше **CHAR** . Цей тип оголошений в **Winnt.h** як показано нижче: #ifdef UNICODE typedef WCHAR TBYTE; #else typedef unsigned char TBYTE;
TCHAR	Якщо визначено як **Unicode** , то тип **WCHAR,** інакше **CHAR** . Цей тип оголошений в **Winnt.h** як показано нижче: #ifdef UNICODE typedef WCHAR TCHAR; #else typedef char TCHAR;
UCHAR	Беззнаковий **CHAR** . Цей тип оголошений в **Windef.h** як показано нижче: typedef unsigned char UCHAR;
UINT	Беззнаковий **INT** . Цей тип оголошений в **Windef.h** як показано нижче: typedef unsigned int UINT;
UINT_PTR	Беззнаковий **INT_PTR** . Цей тип оголошений в **Basetsd.h** як показано нижче: #if defined (_WIN64) typedef unsigned __int64 UINT_PTR; #else typedef unsigned int UINT_PTR;
UINT32	Беззнаковий **INT32** . Цей тип оголошений в **Basetsd.h** як показано нижче: typedef unsigned int UINT32;
UINT64	Беззнаковий **INT64** . Цей тип оголошений в **Basetsd.h** як показано нижче: typedef usigned __int 64 UINT64;
ULONG	Беззнаковий **LONG** . Цей тип оголошений в **Windef.h** як показано нижче: typedef unsigned long ULONG;
ULONGLONG	64-розрядний беззнаковое ціле число. Цей тип оголошений в **Winnt.h** як показано нижче: typedef unsigned __int64 ULONGLONG; #else typedef double ULONGLONG
ULONG_PTR	Беззнаковий **LONG_PTR** . Цей тип оголошений в **Basetsd.h** як показано нижче:

	#if defined (_WIN64) typedef unsigned __int64 ULONG_PTR; #else typedef unsigned long ULONG_PTR;
ULONG32	Беззнаковий **LONG32** . Цей тип оголошений в **Basetsd.h** як показано нижче: typedef unsigned int ULONG32;
ULONG64	Беззнаковий **LONG64** . Цей тип оголошений в **Basetsd.h** як показано нижче: typedef unsigned __int64 ULONG64;
USHORT	Беззнаковий **SHORT** . Цей тип оголошений в **Windef.h** як показано нижче: typedef unsigned short USHORT;
USN	Оновлення числа послідовності (**USN**). Цей тип оголошений в **Winnt.h** як показано нижче: typedef LONGLONG USN;
VOID	Будь-який тип. Цей тип оголошений в **Winnt.h** як показано нижче: #define VOID void
WCHAR	16-бітовий символ **Unicode** . Цей тип оголошений в **Winnt.h** як показано нижче: typedef wchar_t WCHAR;
WINAPI	Угода про виклики для системних функцій. Цей тип оголошений в **Windef.h** як показано нижче: #define WINAPI __stdcall
WORD	16-бітове беззнаковое ціле число. Цей тип оголошений в **Windef.h** як показано нижче: typedef unsigned short WORD;
WPARAM	Параметр повідомлення. Цей тип оголошений в **Windef.h** як показано нижче: typedef UINT_PTR WPARAM;

Розглянемо принципи створення та реалізації віконних додатків за допомогою WinAPI функцій.

Перш за все створимо звичайний консольний додаток в який додаємо наступний код:

```
#include <iostream>
#include <Windows.h>
int main()
{
	return 0;
}
```

Тут головне підключити бібліотеку Windows.h, оскільки всі API функції розташовані якраз в ній.

На наступному кроці нам необхідно зареєструвати структуру в якій буде знаходитись опис нашого головного вікну програми. Це можна зробити за допомогою функції RegisterClassA(&WNDCLASSA). функції RegisterClassA реєструє клас вікна для подальшого використання у викликах функції CreateWindow або CreateWindowEx. Параметром даної функції є вказівник на структуру WNDCLASS.

Структура WNDCLASSA (winuser.h) Містить атрибути класу вікна, зареєстровані функцією RegisterClass.

```
typedef struct tagWNDCLASSA {
  UINT      style;
```

```
  WNDPROC   lpfnWndProc;
  int       cbClsExtra;
  int       cbWndExtra;
  HINSTANCE hInstance;
  HICON     hIcon;
  HCURSOR   hCursor;
  HBRUSH    hbrBackground;
  LPCSTR    lpszMenuName;
  LPCSTR    lpszClassName;
    } WNDCLASSA, *PWNDCLASSA, *NPWNDCLASSA, *LPWNDCLASSA;
```

Параметри структури WNDCLASSA

Параметр	Тип даних	Опис
style	**UINT**	Стиль класу.
lpfnWndProc	**WNDPROC**	Вказівник на процедуру вікна. Для виклику віконної процедури потрібно використовувати функцію CallWindowProc
cbClsExtra	**int**	Кількість додаткових байтів для виділення відповідно до структури віконного класу. Система ініціалізує байти до нуля
cbWndExtra	**int**	Кількість додаткових байтів для виділення після екземпляра вікна. Система ініціалізує байти до нуля. Якщо програма використовує **WNDCLASS** для реєстрації діалогового вікна, створеного за допомогою директиви **CLASS** у файлі ресурсу, вона має встановити цього члена на **DLGWINDOWEXTRA**
hInstance	**HINSTANCE**	Дескриптор екземпляра, що містить процедуру вікна для класу.
hIcon	**HICON**	Дескриптор до значка класу. Цей член повинен бути дескриптором ресурсу піктограм. Якщо цей член **NULL** , система надає піктограму за замовчуванням.
hCursor	**HCURSOR**	Дескриптор курсору класу. Цей член повинен бути дескриптором ресурсу курсору. Якщо цей член є **NULL** , програма повинна явно встановити форму курсора щоразу, коли миша рухається у вікні програми.
hbrBackground	**HBRUSH**	Дескриптор пензля фону класу. Цей елемент може бути ручкою для фізичної пензля, яка використовується для фарбування фону, або це може бути значення кольору. Значення кольору має бути одним із наведених нижче стандартних системних кольорів (до вибраного кольору слід додати значення 1). Якщо задано значення кольору, його потрібно перетворити на один із таких типів **HBRUSH** : • COLOR_ACTIVEBORDER • COLOR_ACTIVECAPTION • COLOR_APPWORKSPACE • COLOR_BACKGROUND • COLOR_BTNFACE • COLOR_BTNSHADOW • COLOR_BTNTEXT • COLOR_CAPTIONTEXT • COLOR_GRAYTEXT • COLOR_HIGHLIGHT • COLOR_HIGHLIGHTTEXT • COLOR_INACTIVEBORDER • COLOR_INACTIVECAPTION • COLOR_MENU • COLOR_MENUTEXT • COLOR_SCROLLBAR • COLOR_WINDOW • COLOR_WINDOWFRAME • COLOR_WINDOWTEXT Система автоматично видаляє фонові пензлі класу, коли клас не реєструється за допомогою UnregisterClass . Додаток не повинен видаляти ці пензлі. Якщо цей учасник має значення **NULL** , програма повинна фарбувати власний фон кожного разу, коли її попросять намалювати в області клієнта. Для того, щоб визначити , чи повинен бути пофарбований фон, додаток може або обробити WM_ERASEBKGND повідомлення або перевірити **fErase** член PAINTSTRUCT структури , заповненої в BeginPaint функції.
lpszMenuName	**LPCTSTR**	Ім'я ресурсу меню класу, оскільки ім'я відображається у файлі ресурсу. Якщо для ідентифікації меню використовується ціле число, використовуйте макрос MAKEINTRESOURCE . Якщо цей член **NULL** , вікна, що належать до цього класу, не мають меню за замовчуванням.

lpszClassName	**LPCTSTR**	Вказівник на рядок з нульовим завершенням або є атомом. Якщо цей параметр є атомом, це повинен бути атом класу, створений за допомогою попереднього виклику функції RegisterClass або RegisterClassEx . Атом повинен бути у нижньому слові **lpszClassName** ; слово старшого порядку має бути нулем. Якщо **lpszClassName** є рядком, він вказує ім'я класу вікна. Ім'я класу може бути будь-яким іменем, зареєстрованим у RegisterClass або RegisterClassEx , або будь-яким із заздалегідь визначених імен класів елементів керування. Максимальна довжина для **lpszClassName** дорівнює 256. Якщо **lpszClassName** більше максимальної довжини, функція RegisterClass вийде з ладу.

!!!!!!!!!!!!!!!!!!!!!!!!!

Ви повинні заповнити структуру відповідними атрибутами класу, перш ніж передавати її функції. Отож для реєстрації нашого класу, що буде відповідати з головне вікно використаємо такий код:

```
WNDCLASSA MyWin;
memset(&MyWin, 0, sizeof(WNDCLASSA));
MyWin.lpszClassName = "My Win";
MyWin.lpfnWndProc = DefWindowProcA;

RegisterClassA(&MyWin);
```

Після цього необхідно створити саме вікно та активувати функцію промальовки вікна на екрані, для цього скористаємось наступним кодом:

```
HWND hwnd_window;
   hwnd_window = CreateWindowA("My Win", "Window Title", WS_OVERLAPPEDWINDOW, 200, 200,
300, 300, NULL, NULL, NULL, NULL);
   ShowWindow(hwnd_window, SW_SHOWNORMAL);
```

Тут використано функції CreateWindowA та ShowWindow.

Функція CreateWindowA Створює спливаюче, дочірнє або самостійне вікно. Вона визначає клас вікна, назву вікна, стиль вікна та (за бажанням) початкове положення та розмір вікна. Функція також вказує батьків або власника вікна, якщо такі є, і меню вікна.

```
void CreateWindowA(
   lpClassName,
   lpWindowName,
   dwStyle,
   x,
   y,
   nWidth,
   nHeight,
   hWndParent,
   hMenu,
   hInstance,
   lpParam
    );
```

Параметр	Тип даних	Опис
lpClassName	**LPCTSTR**	**Нуль** -завершённий рядок або атом класу , створений попереднім викликом до RegisterClass або RegisterClassEx функції. Атом повинен бути в нижньому порядку слова lpClassName ; слово старшого порядку має бути нулем. Якщо lpClassName є рядком, він вказує ім'я класу вікна. Ім'ям класу може бути будь -яке ім'я, зареєстроване в **RegisterClass** або **RegisterClassEx** , за умови, що модуль, який реєструє клас, також є модулем, який створює вікно. Ім'я класу також може бути будь -яким із заздалегідь визначених імен системних класів. Список назв системних класів див. У розділі Зауваження.
lpWindowName	**LPCTSTR**	Назва вікна. Якщо стиль вікна визначає рядок заголовка, заголовок вікна, на який вказує lpWindowName , відображається у рядку заголовка. При використанні **CreateWindow** для створення елементів керування, таких як кнопки, прапорці та статичні елементи

		керування, використовуйте lpWindowName, щоб вказати текст елемента керування. При створенні статичного **елемента** керування зі стилем SS_ICON використовуйте lpWindowName, щоб вказати назву піктограми або ідентифікатор. Щоб вказати ідентифікатор, використовуйте синтаксис "# num ".
dwStyle	**DWORD**	Стиль вікна, що створюється. Цей параметр може бути комбінацією значень стилю вікна. Детальніше про можливі стилі можна прочитати нижче
x	**int**	Початкове горизонтальне положення вікна. Для перекритого або спливаючого вікна параметр x є початковою координатою x верхнього лівого кута вікна в координатах екрана. Для дочірнього вікна, x є й-координата верхнього лівого кута вікна щодо верхнього лівого кута клієнтської області батьківського вікна. Якщо для цього параметра встановлено значення **CW_USEDEFAULT** , система вибирає положення за замовчуванням для верхнього лівого кута вікна та ігнорує параметр y . **CW_USEDEFAULT** діє лише для вікон, що перекриваються; якщо це вказано для спливаючого або дочірнього вікна, параметри x і y встановлюються на нуль.
y	**int**	Початкове вертикальне положення вікна. Для перекритого або спливаючого вікна параметр y- це початкова координата y верхнього лівого кута вікна у координатах екрана. Для дочірнього вікна y -початкова координата y верхнього лівого кута дочірнього вікна щодо лівого верхнього кута клієнтської області батьківського вікна. Для вікна списку y - початкова координата y верхнього лівого кута клієнтської області списку відносно верхнього лівого кута клієнтської області батьківського вікна. Якщо вікно з перекриттям створюється з **набором бітів** стилю **WS_VISIBLE** , а для параметра x встановлено значення **CW_USEDEFAULT** , параметр y визначає, як відображатиметься вікно. Якщо параметр y - **CW_USEDEFAULT** , то менеджер вікон викликає ShowWindow з прапором **SW_SHOW** після створення вікна. Якщо параметр y - це інше значення, то менеджер вікон викликає **ShowWindow** з таким значенням як параметр nCmdShow .
nWidth	**int**	Ширина вікна в одиницях пристрою. Для вікон з перекриттям nWidth - це ширина вікна, у координатах екрана, або **CW_USEDEFAULT** . Якщо nWidth - **CW_USEDEFAULT** , система вибирає ширину та висоту вікна за умовчанням; ширина за замовчуванням поширюється від початкової координати x до правого краю екрана, а висота за умовчанням-від початкової координати y до верхньої частини області значків. **CW_USEDEFAULT** діє лише для вікон, що перекриваються; якщо **CW_USEDEFAULT** вказано для спливаючого або дочірнього вікна, nWidth і nHeight встановлюються на нуль.
nHeight	**int**	Висота вікна у одиницях пристрою. Для перекритих вікон nHeight - це висота вікна в координатах екрана. Якщо для nWidth встановлено значення **CW_USEDEFAULT** , система ігнорує nHeight .
hWndParent	**HWND**	Дескриптор вікна батьків або власника створюваного вікна. Щоб створити дочірнє вікно або власне вікно, надайте дійсний дескриптор вікна. Цей параметр необов'язковий для спливаючих вікон. Щоб створити вікно , що **містить** лише повідомлення , **подайте HWND_MESSAGE** або дескриптор до існуючого вікна , що **містить** лише повідомлення.
hMenu	**HMENU**	Дескриптор меню або вказує ідентифікатор дочірнього вікна залежно від стилю вікна. Для перекритого або спливаючого вікна hMenu визначає меню, яке буде використовуватися з вікном; він може бути **NULL,** якщо потрібно використовувати меню класу. Для дочірнього вікна hMenu вказує ідентифікатор дочірнього вікна, ціле число, яке використовується елементом керування діалогового вікна для сповіщення батьків про події. Додаток визначає ідентифікатор дочірнього вікна; воно має бути унікальним для всіх дочірніх вікон з однаковим батьківським вікном.
hInstance	**HINSTANCE**	Дескриптор екземпляра модуля, який буде пов'язаний з вікном.
lpParam	**LPVOID**	Покажчик на значення , яке буде передано в вікно через CREATESTRUCT структури (**lpCreateParams** члена) , на який вказує LPARAM парам в WM_CREATE повідомленні. Це повідомлення надсилається у створене вікно цією функцією до його повернення. Якщо програма викликає **CreateWindow** для створення вікна клієнта MDI, lpParam має вказати на структуру CLIENTCREATESTRUCT . Якщо вікно клієнта MDI викликає **CreateWindow** для створення дочірнього вікна MDI, lpParam має вказати на структуру MDICREATESTRUCT . lpParam може бути **нульовим,** якщо не потрібні додаткові дані.

Дана функція повертає тип: **HWND**.

Якщо функція успішна, повернене значення є дескриптором нового вікна.

Якщо функція не працює, повертається значення NULL . Щоб отримати розширену інформацію про помилку, необхідно звернутись до GetLastError

У параметрі *lpClassName* можна вказати такі наперед визначені системні класи . Зверніть увагу на відповідні стилі керування, які можна використовувати в параметрі *dwStyle* .

Системний клас	**Значення**
BUTTON	Позначає маленьке прямокутне дочірнє вікно, яке представляє кнопку, яку користувач може натиснути, щоб увімкнути або вимкнути її. Елементи керування кнопками можна використовувати окремо або в

	групах, вони можуть бути позначені або відображатися без тексту. Елементи управління кнопками зазвичай змінюють зовнішній вигляд, коли користувач натискає на них. Для отримання додаткової інформації див. Кнопки Таблицю стилів кнопок, яку можна вказати в параметрі *dwStyle* , див. У розділі Стилі кнопок .
COMBOBOX	Позначає елемент керування, що складається зі списку та поля вибору, подібного до елемента керування редагуванням. При використанні цього стилю програма повинна або відображати вікно списку постійно, або активувати розкривний список. Якщо вікно списку видно, введення символів у поле вибору виділяє перший запис у списку, який відповідає набраним символам. І навпаки, вибір елемента у вікні зі списком відображає виділений текст у полі вибору. Для отримання додаткової інформації див. Розділи Комбо . Таблицю стилів *списків* зі списком, які можна вказати в параметрі *dwStyle* , див. У розділі Стилі комбо -*вікна* .
EDIT	Позначає прямокутне дочірнє вікно, в яке користувач може вводити текст з клавіатури. Користувач вибирає елемент керування та надає йому фокус клавіатури, натиснувши її або перейшовши до неї, натиснувши клавішу TAB. Користувач може вводити текст, коли елемент редагування відображає миготливий курсор; використовуйте мишу для переміщення курсору, вибору символів, які потрібно замінити, або розташування курсору для вставки символів; або за допомогою клавіші BACKSPACE видаліть символи. Докладнішу інформацію див. У розділі Редагування елементів керування . Таблицю стилів керування редагуванням, яку можна вказати в параметрі *dwStyle* , див. У розділі Редагування стилів керування .
LISTBOX	Позначає список рядків символів. Вказуйте цей елемент керування, коли програма повинна подавати список імен, таких як імена файлів, з яких користувач може вибрати. Користувач може вибрати рядок, натиснувши його. Вибраний рядок виділяється, а повідомлення -сповіщення передається до батьківського вікна. Для отримання додаткової інформації див. Списки . Таблицю стилів *вікон* списку, яку можна вказати в параметрі *dwStyle* , див. У розділі Стилі *вікон* списку .
MDICLIENT	Позначає вікно клієнта MDI. Це вікно отримує повідомлення, які керують дочірніми вікнами програми MDI. Рекомендовані біти стилю - **WS_CLIPCHILDREN** та **WS_CHILD** . Вкажіть стилі **WS_HSCROLL** та **WS_VSCROLL,** щоб створити вікно клієнта MDI, що дозволяє користувачеві прокручувати дочірні вікна MDI у вікні перегляду. Для отримання додаткової інформації див. Інтерфейс кількох документів .
RichEdit	Позначає елемент керування Microsoft Rich Edit 1.0. Це вікно дозволяє користувачеві переглядати та редагувати текст із форматуванням символів та абзаців та може включати вбудовані об'єкти компонентної об'єктної моделі (COM). Докладнішу інформацію див. У розділі Елементи керування багатим редагуванням . Таблицю стилів керування з розширеним редагуванням, яку можна вказати в параметрі *dwStyle* , див. У розділі Багате редагування стилів керування .
RICHEDIT_CLASS	Позначає елемент керування Microsoft Rich Edit 2.0. Цей елемент керування дозволяє користувачеві переглядати та редагувати текст із форматуванням символів та абзаців, а також може включати вбудовані об'єкти COM. Докладнішу інформацію див. У розділі Елементи керування багатим редагуванням . Таблицю стилів керування з розширеним редагуванням, яку можна вказати в параметрі *dwStyle* , див. У розділі Багате редагування стилів керування .
SCROLLBAR	Позначає прямокутник, який містить поле прокрутки та має стрілки напрямку на обох кінцях. Смуга прокрутки надсилає повідомлення сповіщення до свого батьківського вікна кожного разу, коли користувач натискає елемент керування. Батьківське вікно відповідає за оновлення положення поля прокрутки, якщо це необхідно. Для отримання додаткової інформації див. Смуги прокрутки . Таблицю стилів керування смугою прокрутки, яку можна вказати в параметрі *dwStyle* , див. У розділі Стилі керування *смугою* прокрутки .
STATIC	Позначає просте текстове поле, поле або прямокутник, що використовується для позначення, позначення або окремих інших елементів керування. Статичні елементи керування не беруть введення і не забезпечують виведення. Для отримання додаткової інформації див. Статичні елементи керування .

При створенні вікна його стиль задається в якості одного з параметрів функції CreateWindow().

Вікна, що перекриваються, (Overlapped windows). Overlapped window - це вікно верхнього рівня, що має заголовок, границю, і клієнтську область. Вікна цього типу призначені для використання як головне вікно додатка.

Вікно типу WS_OVERLAPPED має тільки заголовок і границю

Вікно типу WS_OVERLAPPEDWINDOW додатково має системне меню, кнопки мінімізації і максимізації.

Спливаючі вікна (Pop-up windows). Pop-up window - це спеціальний тип overlapped window, використовуваний для створення вікон діалогу, вікон повідомлень і інших тимчасових вікон, що відображаються поза клієнтською областю головного вікна додатка.

Для породження спливаючого вікна використовується стиль WS_POPUP. Цей стиль може комбінуватися з WS_CAPTION, для наділення вікна заголовком. Стиль WS_POPUPWINDOW використовується для наділення спливаючого вікна додатково системним меню і границею.

Дочірні вікна (Child windows). Для породження дочірнього вікна використовується стиль WS_CHILD. Дочірнє вікно відображається в межах клієнтської області батьківського. Дочірнє вікно за замовчуванням має тільки клієнтську область. Додатково можна задати будь-які елементи крім меню. Дії, вироблені з батьківським вікном спричиняють дії, вироблені з дочірнім

Батьківське вікно	Дочірнє
Знищується	Знищується перед батьківським
Ховається	Ховається перед батьківським
Переміщається	Переміщається разом з батьківським
Відображається	Відображається після батьківського

Дочірнє вікно не вирізує автоматично з клієнтської області батьківського при отрисовці. Це означає, що батьківське вікно малює поверх дочірнього. Якщо необхідно уникнути цього, батьківське вікно повинне мати стиль WS_CLIPCHILDREN. Аналогічно, щоб уникнути отрисовці з боку братів, дочірнє вікно повинне мати стиль WS_CLIPSIBLINGS.

Нижче наведено стилі вікон. Після створення вікна ці стилі не можна змінювати, за винятком зазначеного.

Константа/значення	**Опис**
WS_BORDER 0x00800000L	Вікно має тонку лінію.
WS_CAPTION 0x00C00000L	Вікно має рядок заголовка (включає стиль **WS_BORDER**).
WS_CHILD 0x40000000L	Вікно - це дочірнє вікно. Вікно з таким стилем не може мати рядка меню. Цей стиль не можна використовувати зі стилем **WS_POPUP** .
WS_CHILDWINDOW 0x40000000L	Те саме, що стиль **WS_CHILD** .
WS_CLIPCHILDREN 0x02000000L	Виключає площу, яку займають дочірні вікна, коли малювання відбувається у батьківському вікні. Цей стиль використовується при створенні батьківського вікна.
WS_CLIPSIBLINGS 0x04000000L	Обрізає дочірні вікна один щодо одного; тобто, коли конкретне вікно дитини отримує **WM_PAINT** повідомлення, в **стилі WS_CLIPSIBLINGS** кліпи стилю всі інші перекривання дочірні вікна з області дочірнього вікна будуть оновлені. Якщо **WS_CLIPSIBLINGS** не вказано, а дочірні вікна перекриваються, під час малювання в клієнтській області дочірнього вікна можна малювати в клієнтській області сусіднього дочірнього вікна.
WS_DISABLED 0x08000000L	Вікно спочатку вимкнено. Відключене вікно не може приймати вхідні дані від користувача. Щоб змінити це після створення вікна, скористайтеся функцією **EnableWindow** .
WS_DLGFRAME 0x00400000L	Вікно має рамку стилю, який зазвичай використовується з діалоговими вікнами. Вікно з таким стилем не може мати рядка заголовка.
WS_GROUP 0x00020000L	Вікно є першим елементом керування групи елементів керування. Група складається з цього першого елемента керування та всіх елементів керування, визначених після нього, аж до наступного **елемента** керування зі стилем **WS_GROUP** . Перший елемент керування в кожній групі зазвичай має стиль **WS_TABSTOP,** щоб користувач міг переходити від групи до групи. Згодом користувач може змінити фокус клавіатури з одного елемента керування в групі на наступний елемент керування в групі за допомогою клавіш напрямку. Ви можете вмикати та вимикати цей стиль, щоб змінити навігацію у діалоговому вікні. Щоб змінити цей стиль після створення вікна, скористайтеся функцією **SetWindowLong** .
WS_HSCROLL 0x00100000L	Вікно має горизонтальну смугу прокрутки.
WS_ICONIC 0x20000000L	Вікно спочатку згорнуто. Те саме, що стиль **WS_MINIMIZE** .
WS_MAXIMIZE 0x01000000L	Вікно спочатку розгорнуто.
WS_MAXIMIZEBOX 0x00010000L	У вікні є кнопка розгортання. Не можна поєднувати зі стилем **WS_EX_CONTEXTHELP** . **WS_SYSMENU** стиль також повинен бути вказаний.
WS_MINIMIZE 0x20000000L	Вікно спочатку згорнуто. Те саме, що стиль **WS_ICONIC** .

WS_MINIMIZEBOX 0x00020000L	У вікні є кнопка згортання. Не можна поєднувати зі стилем **WS_EX_CONTEXTHELP** . **WS_SYSMENU** стиль також повинен бути вказаний.
WS_OVERLAPPED 0x00000000L	Вікно - це вікно з перекриттям. Перекрите вікно має рядок заголовка та рамку. Те саме, що стиль **WS_TILED** .
WS_OVERLAPPEDWINDOW (WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_THICKFRAME | WS_MINIMIZEBOX | WS_MAXIMIZEBOX)	Вікно - це вікно з перекриттям. Те саме, що стиль **WS_TILEDWINDOW** .
WS_POPUP 0x80000000L	Вікно-це спливаюче вікно. Цей стиль не можна використовувати зі стилем **WS_CHILD** .
WS_POPUPWINDOW (WS_POPUP | WS_BORDER | WS_SYSMENU)	Вікно-це спливаюче вікно. У **WS_CAPTION** і **WS_POPUPWINDOW** стилі повинні бути об'єднані , щоб зробити меню вікна видимим.
WS_SIZEBOX 0x00040000L	Вікно має рамку розміру. Те саме, що стиль **WS_THICKFRAME** .
WS_SYSMENU 0x00080000L	У рядку заголовка вікна є меню вікна. **WS_CAPTION** стиль також повинен бути вказаний.
WS_TABSTOP 0x00010000L	Вікно - це елемент керування, який може отримувати фокус клавіатури, коли користувач натискає клавішу TAB. Натискання клавіші TAB змінює фокусування клавіатури на наступний елемент управління зі стилем **WS_TABSTOP** . Ви можете вмикати та вимикати цей стиль, щоб змінити навігацію у діалоговому вікні. Щоб змінити цей стиль після створення вікна, скористайтеся функцією **SetWindowLong** . Щоб створені користувачем вікна та безмодові діалоги працювали з зупинками вкладок, змініть цикл повідомлень, щоб викликати функцію **IsDialogMessage** .
WS_THICKFRAME 0x00040000L	Вікно має рамку розміру. Те саме, що стиль **WS_SIZEBOX** .
WS_TILED 0x00000000L	Вікно - це вікно з перекриттям. Перекрите вікно має рядок заголовка та рамку. Те саме, що стиль **WS_OVERLAPPED** .
WS_TILEDWINDOW (WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_THICKFRAME | WS_MINIMIZEBOX | WS_MAXIMIZEBOX)	Вікно - це вікно з перекриттям. Те саме, що стиль **WS_OVERLAPPEDWINDOW** .
WS_VISIBLE 0x10000000L	Вікно спочатку видно. Цей стиль можна **вмикати** та вимикати за допомогою функції **ShowWindow** або **SetWindowPos** .
WS_VSCROLL 0x00200000L	Вікно має вертикальну смугу прокрутки.

Функція ShowWindow - встановлює стан показу зазначеного вікна.

```
BOOL ShowWindow(
  HWND hWnd,
  int  nCmdShow
    );
```

Параметр	Тип даних	Опис
hWnd	**HWND**	Вказівник до вікна
nCmdShow	**int**	Керує відображенням вікна. Цей параметр ігнорується під час першого запуску програми ShowWindow , якщо програма, яка запустила програму, надає структуру STARTUPINFO.

У наступних викликах цей параметр може мати одне з наступних значень.

Цінність	**Значення**
SW_HIDE 0	Приховує вікно та активує інше вікно.
SW_SHOWNORMAL **SW_NORMAL** 1	Активує та відображає вікно. Якщо вікно мінімізовано або розгорнуто, система поверне його до початкового розміру та положення. Додаток повинен вказати цей прапор під час першого відображення вікна.
SW_SHOWMINIMIZED 2	Активує вікно та відображає його як мінімізоване вікно.

SW_SHOWMAXIMIZED **SW_MAXIMIZE** 3	Активує вікно та відображає його як розгорнуте вікно.
SW_SHOWNOACTIVATE 4	Відображає вікно з останнім розміром і положенням. Це значення схоже на **SW_SHOWNORMAL** , за винятком того, що вікно не активовано.
SW_SHOW 5	Активує вікно та відображає його у його поточному розмірі та положенні.
SW_MINIMIZE 6	Мінімізує зазначене вікно та активує наступне вікно верхнього рівня у порядку Z.
SW_SHOWMINNOACTIVE 7	Відображає вікно як мінімізоване вікно. Це значення схоже на **SW_SHOWMINIMIZED** , за винятком того, що вікно не активовано.
SW_SHOWNA 8	Відображає вікно у його поточному розмірі та положенні. Це значення схоже на **SW_SHOW** , за винятком того, що вікно не активовано.
SW_RESTORE 9	Активує та відображає вікно. Якщо вікно мінімізовано або розгорнуто, система поверне його до початкового розміру та положення. Додаток повинен вказати цей прапор під час відновлення мінімізованого вікна.
SW_SHOWDEFAULT 10	Встановлює стан показу на основі значення **SW_,** зазначеного в структурі STARTUPINFO, переданого функції CreateProcess програмою, яка запустила програму.
SW_FORCEMINIMIZE 11	Мінімізує вікно, навіть якщо потік, якому належить вікно, не відповідає. Цей прапор слід використовувати лише при згортанні вікон з іншого потоку.

Для того, щоб створене вікно реагувало на дії користувача його необхідно підключити до системи аналізу повідомлень, що створюються під час будь яких дій користувача або зовнішніх пристроїв. Для цього скористаємось функцією DispatchMessageA(&MSG).

DispatchMessageA(&MSG) -надсилає повідомлення до віконної процедури. Зазвичай вона використовується для відправки повідомлення, отриманого функцією GetMessage()

```
LRESULT DispatchMessageA(
  const MSG *lpMsg
    );
```

Для того щоб функція DispatchMessageA(&MSG) змогла передати повідомлення програмі його необхідно отримати з потоку повідомлень, для цього використовується функція GetMessage().

GetMessage() - отримує повідомлення з черги повідомлень потоку виклику. Функція розсилає вхідні надіслані повідомлення, доки опубліковане повідомлення не стане доступним для пошуку.

```
BOOL GetMessage(
  LPMSG lpMsg,
  HWND  hWnd,
  UINT  wMsgFilterMin,
  UINT  wMsgFilterMax
    );
```

Параметр	Тип даних	Опис
lpMsg	**LPMSG**	Вказівник на структуру MSG, яка отримує інформацію про повідомлення з черги повідомлень потоку.
hWnd	**HWND**	Дескриптор вікна, повідомлення якого потрібно отримати. Вікно має належати поточному потоку. Якщо *hWnd* має значення **NULL** , **GetMessage** отримує повідомлення для будь **-якого** вікна, що належить поточній нитці, та будь **-яких** повідомлень у черзі повідомлень поточного потоку, значення **hwnd** яких є **NULL** (див. Структуру MSG). Тому, якщо hWnd має значення **NULL** , обробляються як повідомлення вікон, так і повідомлення потоків. Якщо *hWnd* дорівнює -1, **GetMessage** отримує лише повідомлення в черзі повідомлень поточного потоку, значення **hwnd** якого **NULL** , тобто повідомлення потоків, опубліковані PostMessage (коли параметр *hWnd* **NULL**) або PostThreadMessage .
wMsgFilterMin	**UINT**	Ціле значення найменшого значення повідомлення, яке потрібно отримати. Використовуйте **WM_KEYFIRST** (0x0100), щоб вказати перше повідомлення з клавіатури, або **WM_MOUSEFIRST** (**0x0200**), щоб вказати перше повідомлення миші. Використовуйте WM_INPUT тут та у *wMsgFilterMax,* щоб вказати лише повідомлення **WM_INPUT** .

		Якщо і wMsgFilterMin, і *wMsgFilterMax* дорівнюють нулю, **GetMessage** повертає всі доступні повідомлення (тобто фільтрація діапазону не виконується).
wMsgFilterMax	**UINT**	Ціле значення найвищого значення повідомлення, яке потрібно отримати. Використовуйте **WM_KEYLAST,** щоб вказати останнє повідомлення з клавіатури, або **WM_MOUSELAST,** щоб вказати останнє повідомлення миші. Використовуйте WM_INPUT тут та у *wMsgFilterMin,* щоб вказати лише повідомлення **WM_INPUT** . *Якщо і wMsgFilterMin,* і *wMsgFilterMax* дорівнюють нулю, **GetMessage** повертає всі доступні повідомлення (тобто фільтрація діапазону не виконується).

Для додавання додаткового елемента на головне вікно програми необхідно, створити нове вікно але його параметри будуть відрізнятись в залежності від необхідності. Відповідні параметри будуть вказуватись у відповідних полях. Наприклад для створення кнопки необхідно створити вікно з такими параметрами

```
    HWND hwnd_button;
       hwnd_button = CreateWindowA("button", "exit", WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON,
50, 50, 60, 60, hwnd_window, (HMENU) button_exit, NULL, NULL);
```

При цьому слід зауважити що клас вікна обирається "button", оскільки це є стандартним типом вікна для відображення кнопок на вікнах додатків. Для створення інших стандартних елементів віконних додатків: перемикачів, кнопок, полів вводу тощо необхідно використати відповідні типи класів, які перелічені вище.

При внесенні початкових координат слід врахувати, що вони будуть вираховуватись відносно головного вікна програми, а не відносно робочого столу.

В результаті написання програмного коду буде створене вікно програми на якому буде розташована кнопка. Оскільки при створення вікна був вказаний стандартний обробник подій DefWindowProcA.

```
    MyWin.lpfnWndProc = DefWindowProcA;
```

То дане вікно буде виконувати невеликий об’єм функцій: згортатись, розгортатись, переміщуватись, на кнопці буде виконуватись анімація натискування тощо. Для того щоб усі елементи функціонувати у повному обсязі необхідно прописати логіку реакції на кожну подію.

```
#include <iostream>
#include <Windows.h>

int main()
{
	WNDCLASSA MyWin;
	memset(&MyWin, 0, sizeof(WNDCLASSA));
	MyWin.lpszClassName = "My Win";
	MyWin.lpfnWndProc = DefWindowProcA;

	RegisterClassA(&MyWin);

	HWND hwnd_window;
	hwnd_window = CreateWindowA("My Win", "Window Title", WS_OVERLAPPEDWINDOW, 200, 200,
300, 300, NULL, NULL, NULL, NULL);
	ShowWindow(hwnd_window, SW_SHOWNORMAL);

	HWND hwnd_button;
	hwnd_button = CreateWindowA("button", "exit", WS_VISIBLE | WS_CHILD |
BS_DEFPUSHBUTTON, 50, 50, 60, 60, hwnd_window, NULL, NULL, NULL);

	MSG msg;
	while (GetMessageA(&msg, NULL, 0, 0))
	{
		DispatchMessageA(&msg);
		//cout << "Message = " << msg.message << "  Code = "<< msg.wParam << endl;
```

```
    }

    return 0;
}
```

2. Структура звіту лабораторної роботи.

- Титульна сторінка.
- Відповіді на контрольні запитання.
- Програмний код розв’язку індивідуального завдання.
- Скріншоти роботи програмного додатку.
- Висновки.

3. Контрольні запитання

1. Дайте визначення терміну «системне програмне забезпечення»
2. Перелічіть завдання системного програмування
3. Назвіть основні режими роботи ОС Windows 10
4. Що таке API?
5. Особливості Win API
6. Перерахуйте основні етапи створення віконного програмного додатку.

4. Завдання для індивідуального виконання

Створити віконний програмний додаток та розмістити на ньому не менше 5 графічних елементи. Тип та місце розташування продумати з таким розрахунком, щоб у подальшому можна було реалізувати логіку взаємодії між даними елементами. Уникайте повторення дизайну та набору елементів з вашими колегами.

Лабораторна робота 2

Тема: Створення та обробка повідомлень в ОС Windows.

Мета: Ознайомитися з структурою та принципами створення, перехоплення та обробки повідомлень в ОС Windows.

2. Теоретичні відомості

На відміну від традиційних “консольних” додатків, Windows додаток ніколи не очікує введення шляхом виклику спеціальних функцій типу getchar(). Операційна система сама здійснює введення і приймає рішення, якому з вікон він призначений.

За кожним вікном закріплена спеціальна функція, так називана процедура вікна. Для повідомлення вікна про призначеному йому чи введенні про яку-небудь іншу подію Windows викликає цю процедуру і передає їй як параметр код події й інших параметрів. Процедура повинна зробити необхідні дії, що відповідають даній події, і повернути керування операційній системі.

Виклик функції вікна називають передачею повідомлення цьому вікну. Як обов'язковий параметр у функцію вікна передається іменована константа, називана кодом повідомлення (message identifier)

Інші параметри визначають додаткові дані, зв'язані з даним повідомленням. Наприклад, при натисканні лівої кнопки миші в той момент, коли покажчик миші знаходиться над клієнтською областю вікна, вікну передається повідомлення WM_LBUTTONDOWN. При цьому як параметри передаються координати покажчика і стан інших кнопок миші і клавіш SHIFT і CTRL.

Windows використовує два способи для передачі повідомлень вікну:

- постановка повідомлення в чергу повідомлень;
- безпосередній виклик процедури вікна.

1. Черга повідомлень потоку

Кожен потік має свою чергу повідомлень. Потік повинний переглядати свою чергу повідомлень і направляти повідомлення на обробку відповідному вікну. Для цей потік повинний організувати цикл обробки повідомлень, що складає з функцій: GetMessage, TranslateMessage, і DispatchMessage.

```
while (GetMessage(&msg, (HWND) NULL, 0, 0)) {
TranslateMessage(&msg);
DispatchMessage(&msg);
}
```

Функція GetMessage() припиняє виконання потоку до одержання їм повідомлення. Функція повертає значення FALSE, якщо отримане повідомлення WM_QUIT. В всіх інших випадках повертається TRUE. Додаток може завершити себе посилкою повідомлення WM_QUIT. Для цього призначена функція

PostQuitMessage(). Звичайно вона викликається у відповідь на одержання головним вікном додатка повідомлення WM_DESTROY.

При натисканні клавіш Windows посилає активному вікну повідомлення WM_KEYDOWN і WM_KEYUP. Ці повідомлення містять деякі віртуальні коди клавіш, а не символи. Для аналізу натискань і породження повідомлень WM_CHAR, що містять символьні коди клавіш, використовується функція TranslateMessage().

Для того, щоб направити повідомлення на обробку відповідному вікну, використовується функція DispatchMessage().

Для постановки повідомлень у чергу використовується функція PostMessage(). Першим параметром цієї функції є описувач вікна, якому передається повідомлення. Система визначає, який потік породив це вікно і ставить повідомлення йому в чергу. Потік, що викликав функцію PostMessage не чекає обробки повідомлення, а продовжує виконуватись. Функцією PostThreadMessage повідомлення ставиться в чергу потоку, але в якості описувача вікна встановлюється NULL. Потік не повинний відправляти такі повідомлення на обробку своїм вікнам. Для цього в циклі обробці повідомлень повинний бути передбачений відповідний аналіз описувача вікна.

2. Безпосередній виклик процедури вікна

Для безпосереднього виклику процедури вікна призначена функція SendMessage(). Потік, що викликав SendMessage() припиняється до завершення обробки повідомлення.

3. Пересилання даних за допомогою повідомлень

Використовуючи власні повідомлення, можна пересилати дані від одного вікна іншому. При цьому як параметри повідомлення, наприклад, можна вказувати адреса області переданих даних і її довжину. Проблема виникає при необхідності зрадити дані іншому процесу, оскільки в даному випадку адреса області даних не має ніякого змісту. Адже в кожного процесу власний адресний простір. Для передачі даних необхідно створити поділювана ділянка пам'яті і використовувати його для передачі даних. Для цього існує спеціальне повідомлення WM_COPYDATA.

Більшість додатків використовує діалогові вікна для запиту у користувача додаткової інформації для виконання будь-яких команд. Наприклад, команда відкриття файлу вимагає вказівки імені файлу, так що додаток створює діалогове вікно, щоб запросити у користувача ім'я файлу. Поки користувач не вкаже ім'я файлу, команда не буде виконана. Після цього програма знищує це діалогове вікно. У цьому випадку використовується модальное діалогове вікно. Інший

приклад: текстовий редактор може використовувати немодальногодіалогове вікно, для команди пошуку. Поки редактор шукає введену фразу, діалогове вікно залишається на екрані. Більш того, користувач може повернутися до редагування тексту, не закриваючи діалог пошуку. Або користувач може ввести іншу фразу для пошуку. Таке діалогове вікно залишається відкритим, поки програма не завершиться або користувач безпосередньо не вибере команду закриття цього діалогу.

Щоб створити діалогове вікно, програма має надати системі шаблон діалогу, що описує зміст і стиль діалогу, і діалогову процедуру. Діалогова процедура виконує приблизно такі ж завдання, що і процедура обробки подій вікна. Діалогові вікна належать до визначених класу вікон. Windows використовує цей клас і відповідну процедуру обробки подій для модальних і немодального діалогів. Ця процедура обробляє одні повідомлення самостійно, а інші передає на обробку діалогової процедури програми. У додатку немає безпосереднього доступу до цього зумовленої класу і відповідної йому процедурі обробки подій. Для зміни стилю і поведінки діалогу програма повинна використовувати шаблон діалогового вікна і діалогову процедуру.

Для створення модального діалогу використовується функція DialogBox, а для створення немодального діалогу - CreateDialog:

int WINAPI DialogBox (HANDLE hInst, LPCSTR template, HWND parent, DLGPROC DlgFunc)

HWND WINAPI CreateDialog (HANDLE hInst, LPCSTR template, HWND parent, DLGPROC DlgFunc)

Параметри: *hInst*- дескриптор екземпляра програми (модуля, в якому знаходиться шаблон); *template*- ім'я ресурсу, що описує діалог; *parent*- дескриптор батьківського вікна; *DlgFunc*- діалогова функція наступного формату:

BOOL CALLBACK DlgFunc (HWND hw, UINT msg, WPARAM wp, LPARAM lp)

Параметри діалогової функції такі ж, як у звичайній функції обробки подій. Відмінність цієї функції - вона викликається з визначеною функції обробки подій для діалогових вікон. Вона повинна повернути значення TRUE, якщо опрацювала передане їй повідомлення, або FALSE в іншому випадку. Вона ні в якому разі не повинна сама викликати DefWindowProc.

При створенні діалогового вікна діалогова процедура отримує повідомлення WM_INITDIALOG. Якщо у відповідь на це повідомлення процедура повертає FALSE, діалог не буде створено: функція DialogBox поверне значення -1, а CreateDialog- NULL.

Модальне діалогове вікно блокує выкно вказане в якості батьківського вікна і з'являється поверх нього (незалежно від стилю WS_VISIBLE). Додаток закриває модальне діалогове вікно за допомогою функції

BOOL WINAPI EndDialog (HWND hw, int result)

Додаток має викликати цю функцію з діалогової процедури у відповідь на повідомлення від кнопок "OK", "Cancel" або команди "Close" з системного меню діалогу. Параметр resultпередається програмі як результат повернення з функції DialogBox.

Немодального діалогове вікно з'являється поверх зазначеного в якості батьківського вікна, але не блокує його. Діалогове вікно залишається поверх батьківського вікна, навіть якщо воно неактивно. Програма сама відповідає за відображення / приховування вікна (за допомогою стилю WS_VISIBLEі функції ShowWindow). Повідомлення для немодального діалогового вікна виявляються в основний черги повідомлень програми. Щоб ці повідомлення були коректно оброблені, слід включити в цикл обробки повідомлень виклик функції:

*BOOL WINAPI IsDialogMessage (HWND hwDlg, MSG * lpMsg)*

Якщо ця функція повернула TRUE, то повідомлення оброблено і його не слід передавати функціям TranslateMessagei DispatchMessage.

Немодального діалогове вікно знищується, якщо знищується його батьківське вікно. У всіх інших випадках програма повинна сама дбати про знищення немодального діалогового вікна, використовуючи виклик:

BOOL WINAPI DestroyWindow (HWND hw)

Функція MessageBox() створює, вказує на екрані і використовує вікно сполучення. Вікно повідомлення містить визначається програмою повідомлення і заголовок, плюс будь-яка комбінація зумовлених значків і командних кнопок.

```
int MessageBox (
   HWND hWnd,
   LPCTSTR lpText,
   LPCTSTR lpCaption,
   UINT uType
);
```

Параметер	Значення
hWnd	Дескриптор вікна власника, яке створює вікно повідомлення. Якщо цей параметр - NULL, то вікно повідомлення не має вікна власника.
lpText	Покажчик на символьний рядок з нулем в кінці, яка містить повідомлення яке показується на екрані.
lpCaption	Покажчик на символьний рядок з нулем в кінці, яка містить заголовок діалогового вікна (вікна повідомлення). Якщо цей параметр - NULL, то використовується заданий за замовчуванням заголовок Error (Помилка).

uType	Встановлює зміст і режим роботи діалогового вікна. Цим параметром може бути комбінація прапорців з нижче перерахованих груп прапорців.

Щоб позначити кнопки, які відображаються на екрані у вікні повідомлення, задайте одне з нижче перерахованих значень.

Значення	Призначення
MB_ABORTRETRYIGNORE	Вікно повідомлення містить три командних кнопки: Припинити (Abort), Поторій (Retry) і Пропустити (Ignore) .
MB_CANCELTRYCONTINUE	Вікно повідомлення містить три командних кнопки: Скасувати (Cancel), Спробувати знову (Try Again), Продовжити (Continue) . Використовуйте цей тип вікна повідомлення замість типу MB_ABORTRETRYIGNORE .
MB_HELP	Додає в вікно повідомлення кнопку Довідка (Help) . Коли користувач клацає по кнопці Довідка (Help) або натискає кнопку F1 , система відправляє власнику повідомлення WM_HELP .
MB_OK	Вікно повідомлення містить одну командну кнопку: OK . Це - значення за замовчуванням.
MB_OKCANCEL	Вікно повідомлення містить дві командних кнопки: OK і Скасувати (Cancel) .
MB_RETRYCANCEL	Вікно повідомлення містить дві командних кнопки: Поторій (Retry) і Скасувати (Cancel) .
MB_YESNO	Вікно повідомлення містить дві командних кнопки: Так (Yes) і Ні (No) .
MB_YESNOCANCEL	Вікно повідомлення містить три командних кнопки: Так (Yes), Ні (No) і Скасувати (Cancel) .

Щоб показати на екрані значок у вікні повідомлення, встановіть одне з нижче перерахованих значень.

значення	призначення
MB_ICONEXCLAMATION	У вікні повідомлення з'являється іконка знака оклику.
MB_ICONWARNING	У вікні повідомлення з'являється іконка знака оклику.
MB_ICONINFORMATION	У вікні повідомлення з'являється значок, що складається з малої літери і в колі.
MB_ICONASTERISK	У вікні повідомлення з'являється значок, що складається з малої літери і в колі.

MB_ICONQUESTION	У вікні повідомлення з'являється іконка знака питання.
MB_ICONSTOP	У вікні повідомлення з'являється значок стоп-сигналу.
MB_ICONERROR	У вікні повідомлення з'являється значок стоп-сигналу.
MB_ICONHAND	У вікні повідомлення з'являється значок стоп-сигналу.

Щоб вказати основну кнопку (за замовчуванням), встановіть одне з нижче перерахованих значень.

значення	призначення
MB_DEFBUTTON1	Перша кнопка - основна (кнопка за замовчуванням). MB_DEFBUTTON1 - значення за замовчуванням, якщо MB_DEFBUTTON2 , MB_DEFBUTTON3 , або MB_DEFBUTTON4 не визначені.
MB_DEFBUTTON2	Друга кнопка - основна кнопка.
MB_DEFBUTTON3	Третя кнопка - основна кнопка.
MB_DEFBUTTON4	Четверта кнопка - основна кнопка.

Якщо вікно повідомлення має кнопку Відмінити (Cancel), то функція повертає значення IDCANCEL, якщо або обробляється клавіша ESC , або обрана кнопка Скасувати (Cancel) . Якщо вікно повідомлення не має кнопки Скасувати (Cancel), натискання ESC не має ніякої дії.

Якщо функція завершується помилкою, яке значення дорівнює нулю. Щоб отримати додаткову інформацію про помилку, викличте GetLastError .

Якщо функція завершується успішно, повертається значення - одне з нижче перерахованих значень пункту меню.

значення	призначення
IDABORT	Була обрана кнопка Припинити (Abort) .
IDCANCEL	Була обрана кнопка Скасувати (Cancel) .
IDCONTINUE	Була обрана кнопка Продовжити (Continue) .
IDIGNORE	Була обрана кнопка Пропустити (Ignore).
IDNO	Була обрана кнопка Ні (No) .
IDOK	Була обрана кнопка OK .
IDRETRY	Була обрана кнопка Повтор (Retry) .

IDTRYAGAIN	Була обрана кнопка Спробувати знову (Try Again).
IDYES	Була обрана кнопка Так (Yes) .

Наприклад наступний код виведе на екран повідомлення у вигляді стандартного віконечка windows:

```
MessageBox(NULL, L"This is test message!!!", L"This is box caption", MB_OK);
```


```
#include <iostream>
#include <string>
#include <Windows.h>

enum elements_id {
button_exit = 1,
button_info,
edit_1,
static_1,
combobox_1
};
HWND hwnd_main_window;// дескриптор головного вікна
HWND hwnd_button_exit_unicode;// дескриптор кнопки виходу
HWND hwnd_button_info_macros;// дескриптор кнопки виводу інформації приклад діалогового
вікна
HWND hwnd_edit_1;// дескриптор для поля вводу/виводу/редагування інформації
HWND hwnd_static_1;// дескриптор для поля виводу інформації
HWND hwnd_combobox_1; // дескриптор для випадаючого вікна

//додаткова функція, що буде додавати елемент до випадаючого вікна
//дана функція носить демонстаційних характер
void add_element_in_dropbox(HWND hwnd, const char* str)
{
	SendMessageA(hwnd, CB_ADDSTRING, NULL, (LPARAM) str);
    }


using namespace std;
LRESULT WINAPI MyProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)
{
	switch (Msg)
	{
	case WM_CREATE: {//подія яка виникає при створенні вікна проте саме вікно ще не
промалювалось
		// наприклад можна перевірити чи виконуються деякі стартові умови
		break;
	}
	case WM_DESTROY: // подія коли нажатий хрестик закриття програми
	{
		PostQuitMessage(0);
```

```
			break;
		}
		case WM_KEYDOWN : // подія коли нажата кнопка на клавіатурі
		{
			cout << "Key press = " << wParam << endl;
			break;
		}
		case WM_LBUTTONDOWN : // подія коли нажата ліва кнопка миші
		{
			cout << "Left Buttom down" << endl;
			break;
		}
		case WM_MOUSEMOVE://подія коли мишка рухається
		{
			UINT posX = LOWORD(lParam);
			UINT posY = HIWORD(lParam);

		//	cout << "MouseX = " << posX << " MouseY = " << posY << endl;
			break;
		};
		case WM_COMMAND :
		{
			//Повідомлення WM_COMMAND відправляється тоді, коли користувач вибирає
командний пункт з меню,
			//коли орган управління відправляє повідомну повідомлення своєму батьківському
вікну,
			//або коли транслюється натискання клавіші - прискорювача.
			//wParam
			//Старше слово визначає код повідомлення, якщо це повідомлення від органу
управління.Якщо повідомлення від
			//клавіші - прискорювача, цей параметр дорівнює 1. Якщо повідомлення від меню,
цей параметр 0.
			//Молодше слово визначає ідентифікатор (ID) пункту меню, органу управління,
або прискорювача клавіатури(акселератора).
			//lParam
			//Дескриптор(адрес) органу управління, що відправляє повідомлення, якщо
повідомлення від органу управління.
			//Інакше, цей параметр має значення NULL.

			if (LOWORD(wParam) == button_exit) // перший перевірки який саме елемент був
активований за його ІДЕНТИФІКАТОРОМ
			{
				PostQuitMessage(0);
			}
			if (hwnd_button_info_macros == (HWND) lParam) // перший перевірки який саме
елемент був активований за його АДРЕСОЮ
			{
				//Функція MessageBox створює, вказує на екрані і використовує вікно
сполучення.Вікно повідомлення містить
				//визначається програмою повідомлення і заголовок, плюс будь - яка
комбінація зумовлених значків і командних кнопок.
				MessageBox(NULL, L"This is test message!!!", L"This is box caption",
MB_OK);
				add_element_in_dropbox(hwnd_combobox_1, "Add_new_str");
				MessageBox(NULL, L"NEW string was add!!!", L"This is box caption",
MB_OK);
			}
			if (LOWORD(wParam) == edit_1)
			{
				//щоб отримати інформацію з вікна типу edit необхідно спочатку вияснити
яка подія відбулась
				//для отримання коду необхідно взяти старше слово параметра wParam
				int temp_code = HIWORD(wParam);
				LPWSTR text_in_edit_1 = new WCHAR[100];//оголосимо тимчасовий буфер де
будемо зберігати значення з поля edit
				// будемо використовувати масив символів unicode ;
```

```
            // щоб побачити різницю між EN_UPDATE та  EN_CHANGE роздукументуйте відповідні
MessageBox-и
                    //для того щоб встановити значення іншого вікна скористаємось наступною
функцією
                    //GetWindowText(дескриптор вікна звідки буде братись текст, стрічка
куди буде записано текст, максимальна кількість считаних символів-1 )
                    if (temp_code == EN_UPDATE) //код події яка виникає після того як текст
було змінено АЛЕ ще не відображено на екрані
                    {
                            GetWindowText(hwnd_edit_1, text_in_edit_1, 100);
                            //MessageBox(NULL, text_in_edit_1, L"Show before change edit",
MB_OK);
                    }
                    if (temp_code == EN_CHANGE) //код події яка виникає після того як текст
було змінено І відображено на екрані
                    {
                            GetWindowText(hwnd_edit_1, text_in_edit_1, 100);
                            //MessageBox(NULL, text_in_edit_1, L"Show after change edit",
MB_OK);
                    }
                    //для того щоб встановити значення іншого вікна скористаємось наступною
функцією
                    //SetWindowText(дескриптор вікна куди буде передаватись текст, текс
який буде переданий)
                    if (temp_code == EN_CHANGE) //код події яка виникає після того як текст
було змінено І відображено на екрані
                    {
                            SetWindowText(hwnd_static_1, text_in_edit_1);
                            //MessageBox(NULL, text_in_edit_1, L"Show after change edit",
MB_OK);
                    }
                    delete[]text_in_edit_1;
            }

            if (LOWORD(wParam) == combobox_1)
            {
                    //щоб отримати інформацію з вікна типу combobox необхідно спочатку
вияснити яка подія відбулась
                    //для отримання коду необхідно взяти старше слово параметра wParam
                    int temp_code = HIWORD(wParam);
                    if (temp_code == CBN_SELCHANGE) //код події яка виникає після того як
було обрано деякий пункт випадаючого меню
                    {
                            //щоб взнати ідентифікатор (в даному випадку номер) обраної
стрічки необхідно послати йому повідомлення з запитом
                            int number_combobox_element = SendMessage(hwnd_combobox_1,
CB_GETCURSEL, NULL, NULL);
                            string temp_index_combobox = to_string(number_combobox_element);
                            MessageBoxA(NULL, temp_index_combobox.c_str(), "You select
element", MB_OK);
                            //для того щоб взнати текстове повідомлення яке було обрано
необхідно знову послати повідомлення вікну combobox
                            // проте тепер з іншими параметрами
                            //SendMessageA(ідентифікатор вікна з якого будемо брати
інформацію,
                            //CB_GETLBTEXT- параметр що повертає текстовий опис обраної
стрічки,
                            //номер стрічки назву якої хочемо повернути,
                            //змінна куди запишемо назву нашої стрічки-елемента випадаючого
списку)

                            char * line_value = new char[100]; // змінна де будемо зберігати
значення текстового значення обраного елементу
                            SendMessageA(hwnd_combobox_1, CB_GETLBTEXT,
number_combobox_element, (LPARAM) line_value);
```

```
                        MessageBoxA(NULL, line_value, "You select element and change main
window caption", MB_OK);
                        //за допомогою функції SetWindowTextA(hwnd,LPCSTR) - можна
встановлювати нові значення текстових параметрів
                        //наприклад такий варіант команди дозволить змінити назву нашого
логовного вікна програми
                        SetWindowTextA(hwnd_main_window, (LPCSTR)line_value);
                        delete[]line_value;
                        //
                }
                LPWSTR text_in_edit_1 = new WCHAR[100];//оголосимо тимчасовий буфер де
будемо зберігати значення з поля edit
                // будемо використовувати масив символів unicode ;
                // щоб побачити різницю між EN_UPDATE та  EN_CHANGE роздукументуйте
відповідні MessageBox-и
                //для того щоб встановити значення іншого вікна скористаємось наступною
функцією
                //GetWindowText(дескриптор вікна звідки буде братись текст, стрічка
куди буде записано текст, максимальна кількість зчитаних символів-1 )

                if (temp_code == EN_CHANGE) //код події яка виникає після того як текст
було змінено І відображено на екрані
                {
                        GetWindowText(hwnd_edit_1, text_in_edit_1, 100);
                        //MessageBox(NULL, text_in_edit_1, L"Show after change edit",
MB_OK);
                }
                //для того щоб встановити значення іншого вікна скористаємось наступною
функцією
                //SetWindowText(дескриптор вікна куди буде передаватись текст, текс
який буде переданий)
                if (temp_code == EN_CHANGE) //код події яка виникає після того як текст
було змінено І відображено на екрані
                {
                        SetWindowText(hwnd_static_1, text_in_edit_1);
                        //MessageBox(NULL, text_in_edit_1, L"Show after change edit",
MB_OK);
                }
                delete[]text_in_edit_1;
        }

        break;
    }
    };

    return DefWindowProcA(hWnd, Msg, wParam, lParam);
}
int main()
{
    WNDCLASSA MyWin;
    memset(&MyWin, 0, sizeof(WNDCLASSA));
    MyWin.lpszClassName = "My Win";
    MyWin.lpfnWndProc = MyProc;

    RegisterClassA(&MyWin);


    //CreateWindow - макрос, що буде обирати між CreateWindowA() та CreateWindowW()
    // по замовчуванню стоїть CreateWindowW()
    hwnd_main_window = CreateWindowA("My Win", "Window Title", WS_OVERLAPPEDWINDOW, 200,
200, 300, 300, NULL, NULL, NULL, NULL);
    ShowWindow(hwnd_main_window, SW_SHOWNORMAL);

    //CreateWindowW - використовує надписи в UNICODE форматі, тому перед використанням
текст в ASCII форматі
```

```
	//треба перетворити в UNICODE за допомогою функції TEXT() або поставивши перед текстом літеру L
	hwnd_button_exit_unicode = CreateWindowA("button", "exit", WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON, 50, 50, 60, 60, hwnd_main_window, (HMENU)button_exit, NULL, NULL);

	//CreateWindowA - використовує надписи в ASCII форматі, тому текст записується просто в подвійних лапках
	hwnd_button_info_macros = CreateWindowA("button", "info", WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON, 110, 50, 60, 60, hwnd_main_window, (HMENU)button_info, NULL, NULL);

	// створення поля для вводу/виводу/редагування символьної інформації - edit
	// в даному прикладі поле буде відцентровано по правому краї (ES_RIGHT) та в нього можна буде ввести тільки цифри (ES_NUMBER)
	hwnd_edit_1 = CreateWindowA("edit", "123", WS_VISIBLE | WS_CHILD | WS_BORDER | ES_RIGHT | ES_NUMBER, 50, 120, 120, 20, hwnd_main_window, (HMENU)edit_1, NULL, NULL);

	// створення поля для виводу символьної інформації - static
	// в даному прикладі поле буде відцентровано по лівому краї (ES_LEFT)
	hwnd_static_1 = CreateWindowA("static", "Number in edit box", WS_VISIBLE | WS_CHILD | ES_LEFT , 50, 150, 120, 40, hwnd_main_window, (HMENU)static_1, NULL, NULL);

	// створення випадаючого списку
    // в даному прикладі поле буде мати можливість випадаючого меню оскільки стоїть параметр (CBS_DROPDOWN)
	//коли встановлюєте висоту(5 параметр з кінця) даного елементу то враховуйте те що саме настільки і буде розгортатись меню!!!!!
	hwnd_combobox_1 = CreateWindowA("combobox", "", WS_VISIBLE | WS_CHILD | CBS_DROPDOWN, 180, 50, 100, 150, hwnd_main_window, (HMENU)combobox_1, NULL, NULL);
	// параметр CB_ADDSTRING - дозволяє додавати в кінець елементи випадаючого меню
	SendMessageA(hwnd_combobox_1, CB_ADDSTRING, 0, (LPARAM)"First line");
	SendMessageA(hwnd_combobox_1, CB_ADDSTRING, 0, (LPARAM)"Second line");
	// параметр CB_SETCURSEL - дозволяє встановлювати елемент по замовчуванню
	SendMessageA(hwnd_combobox_1, CB_SETCURSEL, 1, 0);

		MSG msg;
	while (GetMessage(&msg, NULL, 0, 0))
	{
		TranslateMessage(&msg);
		DispatchMessage(&msg);
		//cout << "Message = " << msg.message << "  Code = "<< msg.wParam << endl;
	}
	return 0;
}
```

2. Структура звіту лабораторної роботи.

- Титульна сторінка.
- Відповіді на контрольні запитання.
- Програмний код розв’язку індивідуального завдання.
- Скріншоти роботи програмного додатку.
- Висновки.

3. Контрольні запитання

7. Дайте визначення терміну «повідомлення» в розумінні системного програмування
8. Опишіть можливості та параметри функції SendMessage()

9. Опишіть події які можуть виникнути під час роботи з вікном типу “combobox”
10. Опишіть події які можуть виникнути під час роботи з вікном типу “edit”.
11. Опишіть події які можуть виникнути під час роботи з вікном типу “scrollbar”
12. Для чого використовується функція DispatchMessage(MSG)?

4. Завдання для індивідуального виконання

Для графічного інтерфейсу, що був спроектовано на попередній лабораторній роботі розробити та реалізувати множину функцій для її активної взаємодії з користувачем та обміном відповідних повідомлень між собою. Уникайте повторення дизайну та набору елементів з вашими колегами.

Лабораторна робота 3

Тема: Процеси та потоки в ОС Windows.

Мета: Ознайомитися з структурою та принципами роботи багатопоточних програмних додатків в ОС Windows.

3. Теоретичні відомості

Робота з процесами та потоками має багато загальних моментів, але й відмінностей у них чимало. В основному ці відмінності криються в самій суті цих понять. Давайте розглянемо для початку кожен із них докладніше.

Процес – це деяка частина роботи ОС, що має унікальний ідентифікаційний номер – id, та адресний простір. Адресний простір – деякий список адрес у пам'яті, з якими відбувається робота цього процесу. З іншими адресами процесу доводиться працювати через системний виклик. Одна програма може включати як кілька процесів, так і один, причому останнє найчастіше використовується. Розбиття на процеси дозволяє розпаралелити завдання, завдяки чому прискорити роботу, але в більшості випадків для цього простіше і вигідніше використовувати потоки, які набагато швидше взаємодіють один з одним і мають низку інших позитивних моментів, що і призвело до меншої використання процесів.

У багатьох завданнях можна виділити ряд підзадач, кожну з яких можливо вирішити або незалежно від інших підзадач, або з їх мінімальною кооперацією. При цьому підзадачі виконуються конкурентно (в однопроцесорній системі) або паралельно в многопроцессорной системі. В багатопотоковій моделі кожна така підзадача існує як індивідуальний потік виконання всередині одного і того ж процесу.

Потік – це частина самого процесу, що виконує певний список дій. У кожного процесу є як мінімум один потік, і їх збільшення забезпечує розпаралелювання процесу. Кожен потік, як частина процесу, має доступ до всього адресного простору процесу, всіх його пристроїв і змінних. Тому взаємодія двох окремих потоків реалізується дуже просто і вимагає звернення до системи. Тому використання потоків більш поширене, ніж процесів.

Сам потік є стеком команд з лічильником, що володіє декількома важливими властивостями, такими як стан і пріоритет. Стани потоку всього три: стан активності, тобто потік виконується на даний момент, стан не активності, коли потік очікує на виділення процесора для переходу в стан активності, і третє – стан блокування, коли потоку не виділяється час (відповідно він не займає місце в черзі, звільняючи ресурси) незалежно від його пріоритету.

Кожна програма створює процес, який в свою чергу породжує як мінімум один потік в якому відбувається обробка даних. Для прикладу розглянемо

приклад звичайної програми, що буде виводити на екран повідомлення про id потоку який був визначений для неї, а також підраховуватиме кількість ітерацій циклу які встигнуть пройти, поки користувач не натисне будь яку кнопку.

```
#include <iostream>
#include <thread>
#include <chrono>
#include <conio.h>

using namespace std;


int main() {
	int CounterIteration = 0;
	while (true) {
		CounterIteration++;
		//метод з простору імен this_thread get_id() повертає id потоку який привязний до програми
		//в даному прикладі процес породжує тільки 1 потік, тому все відбувається послідовно
		cout << "Main theard id -> " << this_thread::get_id() << " Iteration number is = " << CounterIteration <<
endl;
		//метод sleep_for() приймає на вхід час в форматі який задаєтсья за допомогою класу chrono
		//та призупиняє роботу процесу на вижначений проміжок часу
		this_thread::sleep_for(chrono::milliseconds(500));
			if (_kbhit())
				break;
	}
}
```

В результати виконання даної програми можна побачити інформацію про процес, а також скільки разів було виконано тіло циклу.

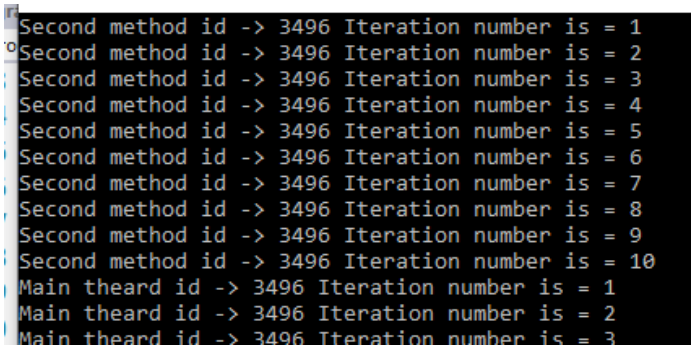
```
Main theard id -> 13580 Iteration number is = 1
Main theard id -> 13580 Iteration number is = 2
Main theard id -> 13580 Iteration number is = 3
Main theard id -> 13580 Iteration number is = 4
Main theard id -> 13580 Iteration number is = 5
Main theard id -> 13580 Iteration number is = 6
Main theard id -> 13580 Iteration number is = 7
Main theard id -> 13580 Iteration number is = 8
```

Якщо модифікувати дану програму та додати в неї ще одну функцію, яка буде з деякою затримної виводити інформаію про свою роботу, при цьому функцію запустити перед головним циклом програми.

```
#include <iostream>
#include <thread>
#include <chrono>
#include <conio.h>

using namespace std;

void SecondMethod(int);

int main() {
	int CounterIteration = 0;
	//додаткова функція
	SecondMethod(10);
	while (true) {
		CounterIteration++;
		//метод з простору імен this_thread get_id() повертає id потоку який привязний до програми
```

```
		//в даному прикладі процес породжує тільки 1 потік, тому все відбувається послідовно
		cout << "Main theard id -> " << this_thread::get_id() << " Iteration number is = " << CounterIteration << endl;
		//метод sleep_for() приймає на вхід час в форматі який задаєтсья за допомогою класу chrono
		//та призупиняє роботу процесу на вижначений проміжок часу
		this_thread::sleep_for(chrono::milliseconds(500));
			if (_kbhit())
				break;
	}
}

void SecondMethod(int counter) {
	for (int i = 0; i < counter; i++)
	{
		cout << "Second method id -> " << this_thread::get_id() << " Iteration number is = " << i+1 << endl;
		this_thread::sleep_for(chrono::milliseconds(500));
	}
}
```

То в результаті роботи програми можна переконатись, що функції працюють в одному потоці та виконуються послідовно одна за одною. Тобто немає жодного виграшу від розпаралелення завдань і частина завдань змушена простоювати в черзі, поки не відпрацюють попередні завдання.



Дана ситуація цілком допустима, якщо це не зменшує швидкодію роботи програм або вирішити неможливо без завершення попередніх етапів. Наприклад, якщо стоїть завдання проаналізувати зображення, то даний етап не можна розпочати без завершення етапу завантаження самого зображення. Проте, якщо у нас вже є завантажене зображення і очікується наступне, то час очікування можна витратити на процес аналізу існуючого зображення, а не просто стояти в очікуванні.

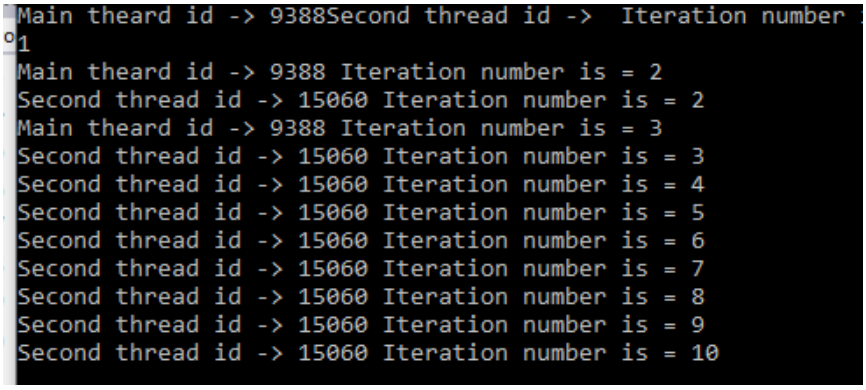
Для створення паралельного процесу осворимо об’єкт класу htread та передамо їй на вхід **вказівник на функцію** (!!!) яку необхідно запустити паралельно. Якщо функція має параметри, то параметри передають через кому після вказівника на функцію. Послідовність та кількість параметрів повинна бути чітко прописана.

```
#include <iostream>
#include <thread>
#include <chrono>
#include <conio.h>

using namespace std;
```

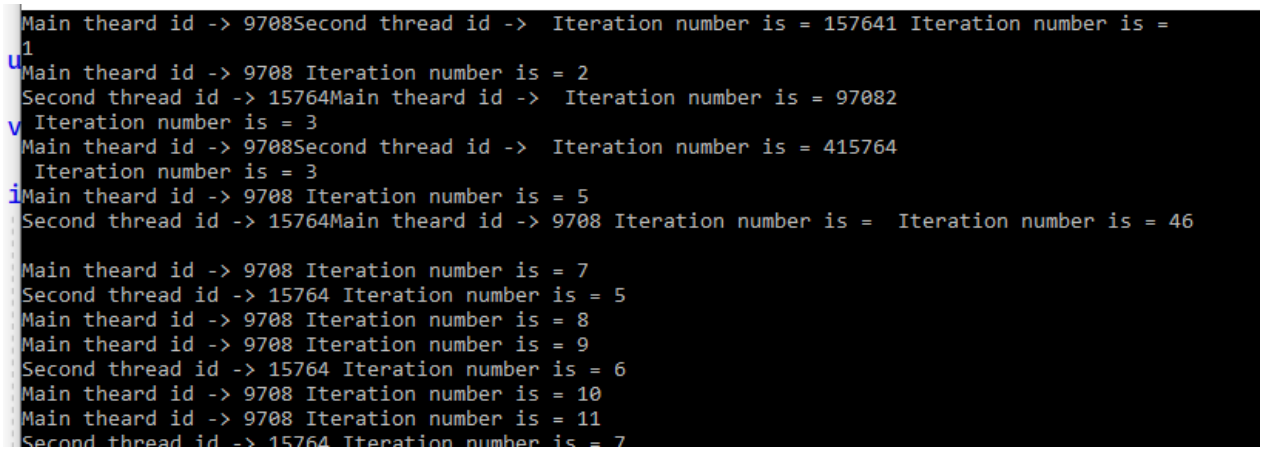
```
void SecondMethod(int);

int main() {
	int CounterIteration = 0;
	//додаткова функція
	thread SecondThread (SecondMethod,10);
	while (true) {
		CounterIteration++;
		//метод з простору імен this_thread get_id() повертає id потоку який привязний до програми
		//в даному прикладі процес породжує тільки 1 потік, тому все відбувається послідовно
		cout << "Main theard id -> " << this_thread::get_id() << " Iteration number is = " << CounterIteration <<
endl;
		//метод sleep_for() приймає на вхід час в форматі який задаєтсья за допомогою класу chrono
		//та призупиняє роботу процесу на вижначений проміжок часу
		this_thread::sleep_for(chrono::milliseconds(500));
			if (_kbhit())
				break;
	}
	//потік треба приєднати до процесу. Приднати можона двома способами
	//по-перше за допомогою join() - в такому випадку програма зупинить виконання свєї роботи
	//і буде чакати поки зупиниться приєднаний потік
	SecondThread.join();
	//по-друге за допомогою потык буде працювати до мометну свого завершенн АБО
	//до моменту коли всі інші потоки завершать свою роботу
	//SecondThread.detach();
}

void SecondMethod(int counter) {
	for (int i = 0; i < counter; i++)
	{
		cout << "Second thread id -> " << this_thread::get_id() << " Iteration number is = " << i+1 << endl;
		this_thread::sleep_for(chrono::milliseconds(1000));
	}
}
```

Для коректної робти потоків необхідно підключити новий потік до вже існуючого. Це можна зробити двомо способами. По-перше за допомгогою метода ***std::thread::join()***. Даний метод блокує поточний потік, доки потік, ідентифікований *this, не завершить своє виконання. Завершення потоку, позначеного *this, синхронізується з відповідним успішним поверненням від join(). Для самого цього *this синхронізація не виконується. Одночасний виклик join() для того самого об’єкта потоку з кількох потоків створює гонку даних, що призводить до невизначеної поведінки.

На прикладі наведено результат роботи програми з підключенням за допомогою метода join(). Як видно з рисунка, головний потік завершив свою роботу, проте оскільки другий ще не допрацював, то програма не завершила свою роботу, а продовжила очікувати повного завершення тругого потоку.



По-друге потік можна підключити за допомгогою метода ***std::thread::detach()***. Даний метод відокремлює потік виконання від об'єкта потоку, дозволяючи продовжувати виконання незалежно. Будь-які виділені ресурси буде звільнено після завершення потоку. Після виклику detach *this більше не володіє жодним потоком. Як видно з рисунку, програма завершила свою роботу в момент завершення головного потоку.



Визначити чи потік був приєднаний можна задопомогою метода ***std::thread::joinable()***. Перевіряє, чи об'єкт std::thread ідентифікує активний потік виконання. Зокрема, повертає true, якщо get_id() != std::thread::id(). Отже, створений за замовчуванням потік не може бути приєднаним. Потік, який завершив виконання коду, але ще не був приєднаний, все ще вважається активним потоком виконання, і тому його можна приєднати.

```
#include <iostream>
#include <thread>
#include <chrono>
#include <conio.h>

using namespace std;

void SecondMethod(int);

int main() {
	int CounterIteration = 0;
```

```
	//додаткова функція
	thread SecondThread;
	cout << boolalpha << "Second thread joinable? : " << SecondThread.joinable();
	SecondThread = std::thread(SecondMethod,10);
	while (true) {
		CounterIteration++;
		//метод з простору імен this_thread get_id() повертає id потоку який привязний до програми
		//в даному прикладі процес породжує тільки 1 потік, тому все відбувається послідовно
		cout << "Main theard id -> " << this_thread::get_id() << " Iteration number is = " << CounterIteration <<
endl;
		//метод sleep_for() приймає на вхід час в форматі який задаєтсья за допомогою класу chrono
		//та призупиняє роботу процесу на вижначений проміжок часу
		this_thread::sleep_for(chrono::milliseconds(500));
			if (_kbhit())
				break;
	}
	//потік треба приєднати до процесу. Приднати можона двома способами
	//по-перше за допомогою join() - в такому випадку програма зупинить виконання свєї роботи
	//і буде чакати поки зупиниться приєднаний потік
	cout << boolalpha << "Second thread joinable? : " << SecondThread.joinable();
	SecondThread.join();
	cout << boolalpha << "Second thread joinable? : " << SecondThread.joinable();
	//по-друге за допомогою потік буде працювати до мометну свого завершенн АБО
	//до моменту коли всі інші потоки завершать свою роботу
	//SecondThread.detach();
	system("pause>>NULL");
}
```

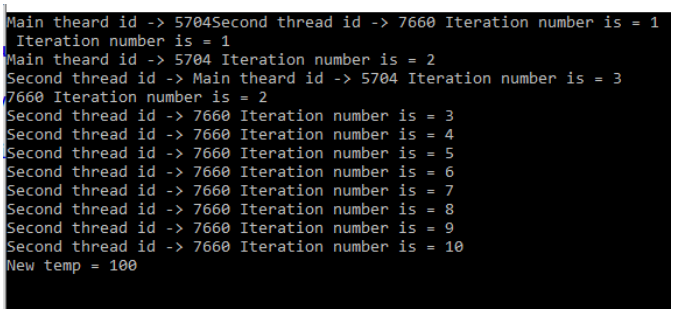
Передача параметрів в потік. Для забезпечення пердачі аргументів в потік використовують пераметри методу який буде запускатись у потоці. Для цього під час приєднання методу до потоку, після вказання назви методу передають його параметри через кому, при цьому послідовність та кількість аргументів повинна співпадати з сигнатурою методу.

```
void SecondMethod(int);

int main() {
    int CounterIteration = 0;
    //додаткова функція
    thread SecondThread (SecondMethod,10);
```

Для того щоб ***отримати значення з потоку***, можна скористатись механізмом посилань при передачі параметрів в функцію. Проте просто ставити амперсанд (&) перед змінною не спрацює, тому що об'єкт thread при передачі цього “не розуміє”, а посилання необхідно описувати як ***std::ref(змінна)***.

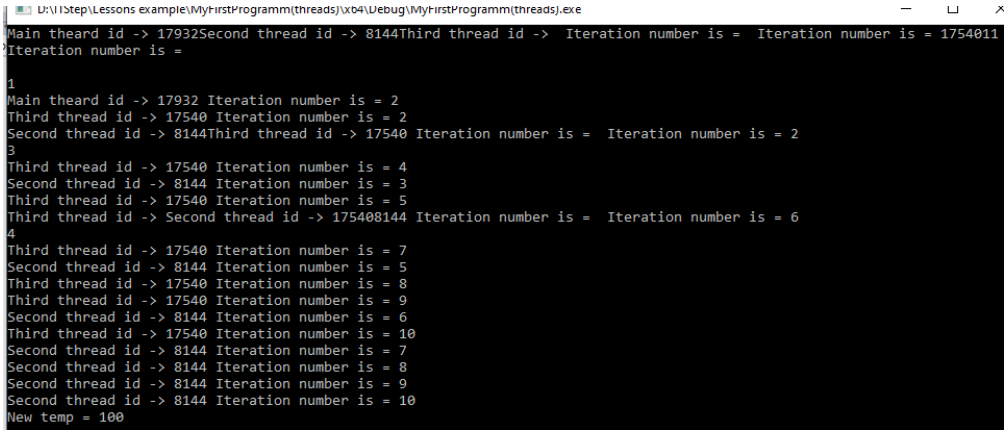
```
void SecondMethod(int&);

int main() {
    int CounterIteration = 0;
    int temp = 10;
    //додаткова функція
    thread SecondThread (SecondMethod,std::ref(temp));
```

Як видно з прикладу, змінна temp була передана по посиланню, і після завершення робти відповідного процесу, можна побачити, що її значення змінилось з 10 на 100.



За анаогією можна запускати і більше потоків.

```
#include <iostream>
#include <thread>
#include <chrono>
#include <conio.h>

using namespace std;

void SecondMethod(int&);
void ThirdMethod(int);

int main() {
	int CounterIteration = 0;
	int temp = 10;
	//додаткова функція
	thread SecondThread (SecondMethod,std::ref(temp));
	thread ThirdThread(ThirdMethod, temp);
	while (true) {
		CounterIteration++;
		cout << "Main theard id -> " << this_thread::get_id() << " Iteration number is = " << CounterIteration <<
endl;
		this_thread::sleep_for(chrono::milliseconds(500));
			if (_kbhit())
				break;
	}
	SecondThread.join();
	ThirdThread.join();
	cout << "New temp = " << temp << endl;
	system("pause>>NULL");
}
void SecondMethod(int &counter) {
	for (int i = 0; i < counter; i++)
	{
		cout << "Second thread id -> " << this_thread::get_id() << " Iteration number is = " << i+1 << endl;
		this_thread::sleep_for(chrono::milliseconds(1000));
	}
	counter = 100;
}
void ThirdMethod(int counter) {
	for (int i = 0; i < counter; i++)
	{
		cout << "Third thread id -> " << this_thread::get_id() << " Iteration number is = " << i + 1 << endl;
		this_thread::sleep_for(chrono::milliseconds(600));
	}
	}
```



Те що в консолі повідомлення “накладаються одне на одне” пояснюється тим що потоки одночасно пробують вивести інформацію на екран, і відповідно накладаються один на одного. Тому треба організовувати правильну синхронізацію потоків.

2. Структура звіту лабораторної роботи.

- Титульна сторінка.
- Відповіді на контрольні запитання.
- Програмний код розв’язку індивідуального завдання.
- Скріншоти роботи програмного додатку.
- Висновки.

3. Контрольні запитання

13. Дайте визначення терміну «процес» в розумінні системного програмування
14. .Що таке потік? У чому його відмінність від процесу?
15. Назвіть переваги використання багатопотоковості.
16. Назвіть недоліки використання багатопотоковості.
17. Вкажіть основні засоби синхронізації потоків.

4. Завдання для індивідуального виконання

Для успішного виконання завдання необхідно розв’язати ряд задач, кожна з яких додає певну кількість балів до підсумкової оцінки:

1) Створити дві функції, які на вхід отримують час своєї роботи (час на який їх треба зупинити, час вибирати випадковим чином в діапазоні від 1 до 5 секунд). В циклі поки не буде нажата клавіша запускати ці функції за допомогою потоків, при цьому під час старту роботи функції повинні вивести на екран повідомлення (“Function 1 2 start work ”/“Function 2 start work”), а після завершення паузи (“Function 2 stop work”/“Function 2 stop work ”). Функцію

можна запускати повторно лише у випадку, якщо вона відпрацювала попередню задачу.

2) Створити масив потоків, кількість яких задає користувач, на вхід кожного з потоків передавати порядковий номер під яким він створювався та час на який він буде призупинений. В циклі запустити ці потоки з деякою затримкою, після чого запущені потоки повинні зупинитись на час переданий як другий параметр, після чого потік повинен вивести на екран повідомлення про свій номер запуску.

3) Створити програму, що буде обчислювати суму в кожному з рядків матриці NxN. Матриця заповнюється впадковими числами. Програма повинна обчислювати суму кожного рядка двома способами:

a. за допомогою функції, яка приймає на вхід двовимірну матрицю та самостійно виводить суму кожного рядка на екран.

b. Набору потоків, які незалежно обчислюють суму кожної з стрічок матриці.

Результати роботи двох підходів порівняти а часовою складністю, для цього використати функцію std::clock() та CLOCKS_PER_SEC. Якщо часові затримки будуть дуже близькими додайте затримки після обробки кожного з рядків, після додавання кожного з елементів, збільшіть розміри масиву. Проаналізуйте чому, при невеликих часових затримках кожної окремої операції додавання, використовувати розпаралелювання не вигідно.

Лабораторна робота 4

Тема: Функції WinAPI отримання та встановлення системних характеристик ОС Windows.

Мета: Отримати практичні навики по використання WinAPI функцій для роботи з системними параметрами в ОС Windows.

4. Теоретичні відомості

Повна інформація про комп'ютер складається з низки даних, які необхідно власноруч збирати, звертаючись до різних інстанцій системи. Така інформація може знадобитися в різних ситуаціях. Наприклад необхідно проаналізувати системні вимоги програм чи додатків та подальшого зіставлення рекомендованих вимог з характеристиками компонентів комп'ютера, що дозволить оцінити можливість правильної роботи програм на ПК.

Для отримання системної інформації можна скористатись WinAPI функціями:

- GetComputerName
- GetSystemMetrics
- GetWindowsDirectoty
- GetKeyboardType
- GetTempPath
- GetSysColor
- GetUserName
- GetSystemDirectory
- GetVersion
- SystemParametersInfo
- GetSystemlnfo
- GetVersioriEx

Більшість з вище наведених функцій мають аналоги для встаовлення тих чи інших параметрів, наприклад:

- SetComputerName
- SetSysColors

Для прикладу розглянемо роботу двох функцій GetComputerName() та GetSystemMetrics().

Функція GetComputerName() повертає NetBios ім'я комп'ютера:

```
BOOL GetComputerName
(
```

LPTSTR lpBuffer, // вказівник на буфер
LPDWORD lpnSize // вказівник на розмір буфера
);

Якщо функція виконається успішно, вона поверне ненульове значення.

Функція GetSystemMetrics повертає системну метрику та системні параметри конфігурації. Системна метрика – це розмір елементів відображення Windows. Усі розміри вказуються у пікселях.

```
int  GetSystemMetrics
   (
  int  nIndex // Затребувана системна метрика або системні установки
  );
```

Параметр *nIndex* Визначає системний показник або налаштування конфігурації, яка буде повернута. Усі значення із префіксом SM_CX визначають ширину елемента, з префіксом SM_CY – висоту. Визначено такі значення:

Значення:	**Опис:**
SM_ARRANGE	Цей прапор визначає, як система впорядковує згорнуті вікна. Для отримання більшої інформації про згорнуті вікна, дивіться секцію "Зауваження".
SM_CLEANBOOT	Визначає тип завантаження системи: 0 – нормальне завантаження; 1- відмовостійке завантаження; 2 - відмовностійке завантаження з мережевою підтримкою. Відмовостійка завантаження (ще вона називається SafeBoot) обходить файли запуску користувача.
SM_CMOUSEBUTTONS	Визначає число кнопок на миші або дорівнює нулю, якщо миша не встановлена.
SM_CXBORDER, SM_CYBORDER	Визначають ширину та висоту, у пікселях, межі вікна. Еквівалентно значенню SM_CXEDGE для вікно з тривимірним переглядом.
SM_CXCURSOR, SM_CYCURSOR	Визначають ширину та висоту, у пікселях, курсору. Система може створювати курсори інших розмірів.
SM_CXDLGFRAME, SM_CYDLGFRAME	Те ж саме, що SM_CXFIXEDFRAME та SM_CYFIXEDFRAME.

SM_CXDOUBLECLK, SM_CYDOUBLECLK	Визначають ширину та висоту прямокутника навколо першого клацання в послідовності подвійного клацання. Друге клацання має відбутися в межах цього прямокутника для визначення двох клацань як одного подвійного клацання. Друге клацання має відбутися в межах цього прямокутника для визначення двох клацань як одного подвійного клацання (два клацання повинні також відбутися в межах зазначеного часу).
SM_CXDRAG, SM_CYDRAG	Ширина і висота, у пікселях, прямокутника, центрованого на точці, що перетягується, для обмеження руху покажчика миші перед стартом операції перетягування. Це дозволяє користувачеві натискати та випускати кнопку миші без ненавмисного старту операції перетягування.
SM_CXEDGE, SM_CYEDGE	Визначають розміри тривимірної межі. Це тривимірні аналоги SM_CXBORDER та SM_CYBORDER.
SM_CXFIXEDFRAME, SM_CYFIXEDFRAME	Товщина рамки навколо периметра вікно, яке має заголовок, але не може змінити розміри. SM_CXFIXEDFRAME – це ширина горизонтального кордону, SM_CYFIXEDFRAME – висота вертикального кордону. Те ж саме, що SM_CXDLGFRAME та SM_CYDLGFRAME
SM_CXFRAME, SM_CYFRAME	Те ж саме, що SM_CXSIZEFRAME та SM_CYSIZEFRAME.
SM_CXFULLSCREEN, SM_CYFULLSCREEN	Визначають ширину та висоту клієнтської області для повноекранного вікна.
SM_CXHSCROLL	Визначає ширину зображення горизонтальної стрілки слайдера.
SM_CYHSCROLL	Визначає висоту горизонтального слайдера у пікселях.
SM_CXHTHUMB	Визначає ширину бігунка горизонтального слайдера.
SM_CXICON, SM_CYICON	Визначають задану за замовчуванням ширину та висоту іконки. Ці значення зазвичай 32x32, але можуть змінюватись залежно від встановлених апаратних засобів дисплея.
SM_CXICONSPACING, SM_CYICONSPACING	Визначають розміри сітки для елементів у вигляді великої іконки. Кожен елемент міститься у прямокутник цього розміру при упорядкуванні іконок. Ці значення завжди завжди більші або рівні SM_CXICON та SM_CYICON.

SM_CXMAXIMIZED, SM_CYMAXIMIZED	Визначають за замовчуванням розміри розгорнутого вікна верхнього рівня.
SM_CXMAXTRACK, SM_CYMAXTRACK	Визначають задані за замовчуванням максимальні розміри вікна, що має заголовок та має можливість змінювати свої межі. Користувач не може зробити рамку вікна більшим за ці розміри. Вікно може скасувати ці значення, обробляючи повідомлення WM_GETMINMAXINFO.
SM_CXMENUCHECK, SM_CYMENUCHECK	Розміри заданого за промовчанням растрового малюнка мітки меню.
SM_CXMENUSIZE, SM_CYMENUSIZE	Визначають розміри кнопок панелі меню, таких як кнопка закриття дочірнього вікна у документі з багатовіконним інтерфейсом (MIDI).
SM_CXMIN, SM_CYMIN	Мінімальна ширина та висота вікна.
SM_CXMINIMIZED, SM_CYMINIMIZED	Визначають розміри нормально згорнутого вікна.
SM_CXMINSPACING, SM_CYMINSPACING	Визначають розміри комірки сітки для згорнутих вікон. Кожне згорнуте вікно поміщається прямокутник цього розміру при упорядкуванні. Ці значення завжди більші або рівні SM_CXMINIMIZED та SM_CYMINIMIZED.
SM_CXMINTRACK, SM_CYMINTRACK	Визначають мінімальні розміри вікна. Користувач не може зробити рамку вікна менше цих розмірів. Вікно може скасувати ці значення, обробляючи повідомлення WM_GETMINMAXINFO.
SM_CXSCREEN, SM_CYSCREEN	Ширина та висота екрана в пікселях.
SM_CXSIZE, SM_CYSIZE	Розміри заголовка вікна чи області заголовка.
SM_CXSIZEFRAME, SM_CYSIZEFRAME	Визначають товщину рамки навколо периметра вікна, що має можливість змінювати розміри. SM_CXSIZEFRAME – ширина горизонтального кордону, і SM_CYSIZEFRAME – висота вертикального кордону. Аналогічно SM_CXFRAME та SM_CYFRAME.

SM_CXSMICON, SM_CYSMICON	Визначають рекомендований розмір невеликої іконки. Маленькі іконки зазвичай з'являються у заголовках вікна.
SM_CXSMSIZE, SM_CYSMSIZE	Визначають розміри невеликих кнопок заголовка.
SM_CXVSCROLL, SM_CYVSCROLL	Визначають розмір зображення стрілки вертикального слайдера.
SM_CYCAPTION	Висота нормальної області заголовка.
SM_CYMENU	Визначає висоту однорядкового меню.
SM_CYSMCAPTION	Висота маленького підпису, у пікселях.
SM_CYVTHUMB	Визначає висоту блоку бігунка у вертикальному слайдері.
SM_DBCSENABLED	Повертається ненульове значення, якщо встановлено двобайтовий набір символів у встановленій версії USER.EXE, інакше нуль.
SM_DEBUG	Повертає ненульове значення, якщо встановлено налагоджувальну версію USER.EXE, інакше нуль.
SM_MENUDROPALIGNMENT	Повертає ненульове значення, якщо меню, що розкривається, вирівняні з правого краю щодо відповідного елемента рядка меню. Нуль, якщо вони вирівняні по лівій межі.
SM_MIDEASTENABLED	Повертає TRUE, якщо система допускає єврейську та арабську мови.
SM_MOUSEPRESENT	Повертає TRUE, якщо мишу встановлено, інакше 0.
SM_MOUSEWHEELPRESENT	Тільки Windows NT: Повертає TRUE, якщо встановлена миша з коліщатком, інакше 0.
SM_NETWORK	У поверненому значенні найменший значний біт буде усунено, якщо мережа присутня; інакше біт очищений. Інші біти зарезервовані для використання.
SM_PENWINDOWS	Повертає TRUE, якщо встановлена робота з пером, інакше 0.

SM_SECURE	Повертає TRUE, якщо захист є.
SM_SHOWSOUNDS	Повертає ненульове значення, якщо користувач потребує програми, що надає інформацію візуально в ситуаціях, де інформація представлена лише у звуковій формі.
SM_SLOWMACHINE	Повертає TRUE, якщо комп'ютер має повільний (low-end) процесор.
SM_SWAPBUTTON	Повертає ненульове значення, якщо значення лівих та правих кнопок миші змінюються.
SM_XVIRTUALSCREEN, SM_YVIRTUALSCREEN	Визначають координати лівої сторони та вершини віртуального екрану. Віртуальний екран – це обмежувальний прямокутник для всіх моніторів дисплея.
SM_CXVIRTUALSCREEN, SM_CYVIRTUALSCREEN	Визначають розміри віртуального дисплея.

Значення, що повертаються:

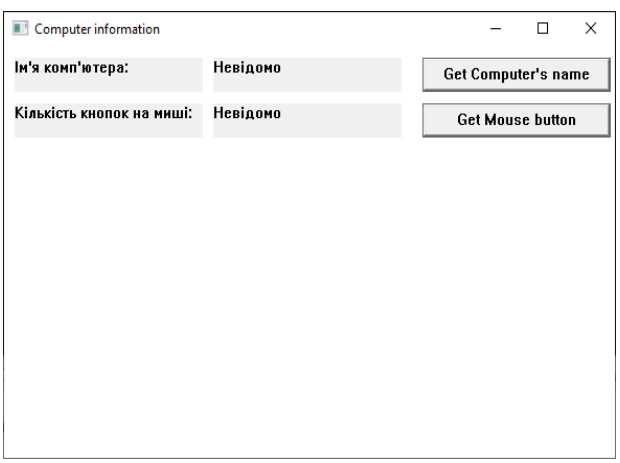
При успіху, функція повертає потрібну системну метрику або конфігураційну установку.

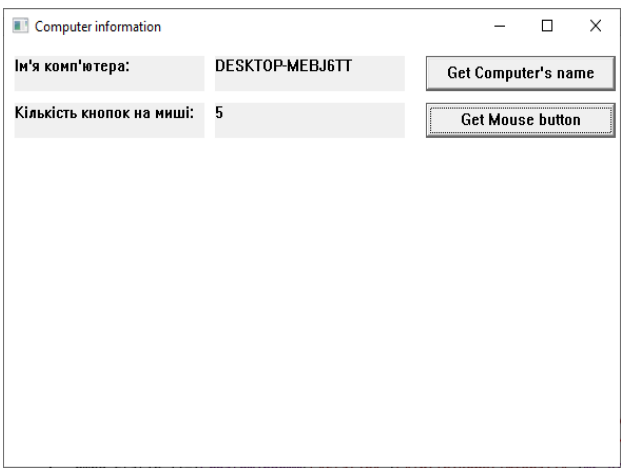
При помилці повертається нуль. Щоб отримати додаткову інформацію про помилку, викличте GetLastError.

```
#include <iostream>
#include <string>
#include <Windows.h>

char* getCompBIOSName();//отримати інформацію про імя компютера
int GetSysCompMetrics(int); //отримати інформацію про системну метрику ОС

enum elements_id {
	button_get_computer_name = 1,
	button_get_mouse_button
};
HWND hwnd_main_window;// дескриптор головного вікна
HWND hwnd_button_get_computer_name;
HWND hwnd_static_1;
HWND hwnd_static_2;
HWND hwnd_button_get_mouse_button;
HWND hwnd_static_3;
HWND hwnd_static_4;


using namespace std;
LRESULT WINAPI MyProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)
{
	switch (Msg)
```

```
	{
	case WM_CREATE: {//подія яка виникає при створенні вікна проте саме вікно ще не промалювалось
		// наприклад можна перевірити чи виконуються деякі стартові умови
		break;
	}
	case WM_DESTROY: // подія коли нажатий хрестик закриття програми
	{
		PostQuitMessage(0);
		break;
	}

	case WM_COMMAND:
	{
		if (LOWORD(wParam) == button_get_computer_name)
		{
			char* compName = getCompBIOSName();
			wchar_t* tempCompName = new wchar_t[strlen(compName) + 1];
			//оскільки треба передавати юнікод, а не char*
			// то використаємо функцію MultiByteToWideChar
			//функція відображає рядок у форматі широких символів (юнікод)
			MultiByteToWideChar(0, 0, compName, strlen(compName), tempCompName,
strlen(compName)+1);
			tempCompName[strlen(compName)] = '\0';
			SetWindowText(hwnd_static_2, tempCompName);
			delete[]tempCompName;
		}

		if (LOWORD(wParam) == button_get_mouse_button)
		{
			wchar_t* tempMoseButton = new wchar_t[100];
			// для перводу цілих чисел в wchar_t можна скористатись функцією wsprintfW
			wsprintfW(tempMoseButton, L"%d", GetSysCompMetrics(SM_CMOUSEBUTTONS));
			SetWindowText(hwnd_static_4, tempMoseButton);
			//або конвертацією через std::to_wstring()
			//SetWindowText(hwnd_static_4,
std::to_wstring(GetSysCompMetrics(SM_CMOUSEBUTTONS)).c_str());
			delete[] tempMoseButton;
		}

		break;
	}
	};

	return DefWindowProcA(hWnd, Msg, wParam, lParam);
}
int main()
{
	WNDCLASSA MyWin;
	memset(&MyWin, 0, sizeof(WNDCLASSA));
	MyWin.lpszClassName = "My Win";
	MyWin.lpfnWndProc = MyProc;

	RegisterClassA(&MyWin);

	hwnd_main_window = CreateWindowA("My Win", "Computer information", WS_OVERLAPPEDWINDOW, 200,
200, 600, 400, NULL, NULL, NULL, NULL);
	ShowWindow(hwnd_main_window, SW_SHOWNORMAL);
	hwnd_button_get_computer_name = CreateWindowA("button", "Get Computer\'s name", WS_VISIBLE |
WS_CHILD | BS_DEFPUSHBUTTON, 400, 10, 180, 30, hwnd_main_window, (HMENU)button_get_computer_name,
NULL, NULL);
	hwnd_static_1 = CreateWindowW(L"static", L"Ім\'я комп\'ютера: ", WS_VISIBLE | WS_CHILD | ES_LEFT, 10,
10, 180, 30, hwnd_main_window, NULL, NULL, NULL);
	hwnd_static_2 = CreateWindowW(L"static", L"Невідомо ", WS_VISIBLE | WS_CHILD | ES_LEFT, 200, 10, 180,
30, hwnd_main_window, NULL, NULL, NULL);
	hwnd_button_get_mouse_button = CreateWindowA("button", "Get Mouse button", WS_VISIBLE | WS_CHILD |
BS_DEFPUSHBUTTON, 400, 50, 180, 30, hwnd_main_window, (HMENU)button_get_mouse_button, NULL, NULL);
```

```
	hwnd_static_3 = CreateWindowW(L"static", L"Кількість кнопок на миші: ", WS_VISIBLE | WS_CHILD | ES_LEFT, 10, 50, 180, 30, hwnd_main_window, NULL, NULL, NULL);
	hwnd_static_4 = CreateWindowW(L"static", L"Невідомо ", WS_VISIBLE | WS_CHILD | ES_LEFT, 200, 50, 180, 30, hwnd_main_window, NULL, NULL, NULL);

	MSG msg;
	while (GetMessage(&msg, NULL, 0, 0))
	{
		TranslateMessage(&msg);
		DispatchMessage(&msg);
	}
	return 0;
}

int GetSysCompMetrics(int metrik) {
	/*int GetSystemMetrics(
		int nIndex		// назва системної метрики або параметра, береться в //довіднику
	);*/
	int rez;
	rez = GetSystemMetrics(metrik);
	return rez;

}
char* getCompBIOSName() {
	//отримати інформацію про імя компютера
/*
BOOL GetComputerNameW(
	LPWSTR  lpBuffer,
	LPDWORD nSize
);
*/
	LPSTR computerName = new char[MAX_COMPUTERNAME_LENGTH + 1];
	LPDWORD t = new DWORD;
	*t = MAX_COMPUTERNAME_LENGTH + 1;
	if (!GetComputerNameA(computerName, t))
		//або з простими типами даних
		//char computerName[MAX_COMPUTERNAME_LENGTH + 1];
		//unsigned long t = MAX_COMPUTERNAME_LENGTH + 1;
		//if (!GetComputerNameA(computerName, &t))
	{
		std::cout << "Error!";
		return 0;
	}
	else
	{
		return computerName;
	}

	}
```





2. Структура звіту лабораторної роботи.

- Титульна сторінка.
- Відповіді на контрольні запитання.
- Програмний код розв’язку індивідуального завдання.
- Скріншоти роботи програмного додатку.
- Висновки.

3. Контрольні запитання

18. Опишіть та дайте визначення для функції GetWindowsDirectoty.
19. Опишіть та дайте визначення для функції GetKeyboardType.
20. Опишіть та дайте визначення для функції GetTempPath.
21. Опишіть та дайте визначення для функції GetSysColor.
22. Опишіть та дайте визначення для функції GetUserName.
23. Опишіть та дайте визначення для функції GetSystemDirectory.
24. Опишіть та дайте визначення для функції GetVersion.
25. Опишіть та дайте визначення для функції SystemParametersInfo.
26. Опишіть та дайте визначення для функції GetSystemlnfo.
27. Опишіть та дайте визначення для функції GetVersioriEx.

4. Завдання для індивідуального виконання

Для успішного виконання завдання необхідно розв’язати ряд задач, кожна з яких додає певну кількість балів до підсумкової оцінки:

Розробити програму з віконним інтерфейсом, що забезпечує отримання наступної системної інформації:

1) Ім'я комп'ютера, ім'я користувача.
2) Шляхи до системних каталогів Windows.
3) Версія ОС.
4) Системні метрики (5 системних метрик на вибір для функції GetSystemMetrics());
5) Системні параметри (5 системних параметри на вибір для функції SystemParametersInfo());
6) Системні кольори (визначити колір для символьних констант та змінити його на будь-який інший (функція GetSysColor)).

Лабораторна робота 5

Тема: Лінійні алгоритми та алгоритми розгалуження з використанням вставок на мові Assembler.

Мета: Отримати практичні навики по реалізації лінійних алгоритмів та алгоритмів розгалуження з використанням вставок на мові Assembler.

5. Теоретичні відомості

Повна інформація про комп'ютер складається з низки даних, які необхідно власноруч збирати, звертаючись до різних інстанцій системи. Така інформація може знадобитися в різних ситуаціях. Наприклад необхідно проаналізувати системні вимоги програм чи додатків та подальшого зіставлення рекомендованих вимог з характеристиками компонентів комп'ютера, що дозволить оцінити можливість правильної роботи програм на ПК.

Асемблер (від англ. assemble - збирати) - компілятор з мови асемблера до команди машинної мови.

Під кожну архітектуру процесора і під кожну ОС чи сімейство ОС існує свій асемблер. Є також так звані кроссассемблери, що дозволяють на машинах з однією архітектурою (або в середовищі однієї ОС) асемблювати програми для іншої цільової архітектури або іншої ОС і отримувати код, що виконується у форматі, придатному до виконання на цільовій архітектурі або в середовищі цільової ОС.

Мова асемблера – тип мови програмування низького рівня. Команди мови асемблера один на один відповідають командам процесора і є зручною символьною формою запису (мнемокод) команд і аргументів. Мова асемблера забезпечує зв'язування частин програми та даних через мітки, що виконується під час асемблювання (для кожної мітки вираховується адреса, після чого кожне входження мітки замінюється на цю адресу).

Кожна модель процесора має свій набір команд та відповідну йому мову (або діалект) асемблера.

Зазвичай програми чи ділянки коду пишуться мовою асемблера у випадках, коли розробнику критично важливо оптимізувати такі параметри, як швидкодія (наприклад, при створенні драйверів) та розмір коду (завантажувальні сектори, програмне забезпечення для мікроконтролерів та процесорів з обмеженими ресурсами, віруси, навісні захисту)).

Синтаксис загальних елементів мови. На відміну від мов програмування високого рівня, де основним елементом мови є оператор, синтаксично програма на асемблері складається з послідовності рядків. Рядок – основна одиниця асемблерної програми.

Синтаксис рядка має такий загальний вигляд:

```
<мітка:> <команда або директива> <операнди> <;кометар>
```

Всі ці чотири поля необов'язкові, у програмі цілком можуть бути і повністю порожні рядки виділення будь-яких блоків коду. Мітка може бути будь-якою комбінацією букв англійського алфавіту, цифр та символів _, $, @, ?, але цифра не може бути першим символом мітки, а символи $ та ? іноді мають спеціальні значення та зазвичай не рекомендуються до використання. Великі та маленькі літери за замовчуванням не відрізняються, але відмінність можна увімкнути, задавши ту чи іншу опцію у командному рядку асемблера. У другому полі, поле команди, може розташовуватися команда процесора, яка транслюється в код, що виконується, або директива, яка не призводить до появи нового коду, а управляє роботою самого асемблера. У полі операнда розташовуються потрібні командою або директивою операнди (тобто не можна вказати операнди і не вказати команду або директиву). І нарешті, у полі коментарів, початок якого відзначається символом ; (крапка з комою), можна написати все, що завгодно - текст від символу «;» до кінця рядка не аналізується асемблером.

Мови високого рівня підтримують можливість вставлення асемблерного коду. Послідовність команд асемблера в С-програмі розміщується в asm-блоці:

```
int main() {
	int A;
	//вставки можна записувати двома варіантами
	//1 спосіб - перед кожною командо. прописувати _asm
	_asm mov A, 20;
	_asm mov eax, A;

	//2 спосіб – об’єднати команди в 1 блок за допомогою {}
	//і перед цим блоком один раз прописати _asm
	_asm {
		add eax, 10;
		mov A, eax;
	}
	cout << A << endl;
	system("pause");
	return 0;
	}
```

Команда MOV копіює дані з операнда-джерела в операнд-одержувач. Вона відноситься до групи команд пересилання даних (data transfer) і використовується у будь-якій програмі. Команда MOV є двомісною (тобто має два операнди): перший операнд визначає одержувача даних (destination), а другий - джерело даних (source):

```
MOV <одержувач>, <джерело>
```

Під час виконання цієї команди змінюється вміст операнда-отримувача, а вміст операнда-джерела не змінюється. Принцип пересилання даних праворуч наліво відповідає прийнятому в операторах присвоєння мов високого рівня, таких як C ++:

```
<одержувач> = <джерело>;
```

```
//!!!!!!!слід пам'ятати про розміри змінних та розміри регістрів
        //! так змінна типу int не поміститься в регастрі ax
        //! 4 байти не розміститься в 2 байтах
        _asm {
                //способи адресації (внесення даних)
                //команда mov a,b - переміщує дані з а до b
                //допустимі переміщення
                //(reg - регістр, mem - память, imm -  число)
                /*
                mov reg, reg
                mov mem, reg
                mov reg, mem
                mov mem, imm
                mov reg, imm
                */
                mov A, eax; // предача значення з регістра
                mov A, 10; // передача числа у десятковому форматі
                mov A, 00000101b; // передача числа у двійковому форматі
                mov A, 11h;// передача числа у шіснадцятковому форматі
                mov A, 010; // передача числа у вісімковому форматі
// типи адресації
                mov ax, bx;//регістрова адресація -> ax стане рівним значенню bx
                mov ax, 20;// безпосередня адресація -> ax стане рівним значенню 20
                mov eax, temp;//пряма адресація -> ax стане рівним значенню змінної temp
}
```

Програмування арифметичних виразів у мові Асемблер відбувається через наступні команди:

add (команда для складання двох чисел): *приймач + джерело = приймач*

adc (команда для складання двох чисел з урахуванням перенесення): *приймач + джерело + CF = приймач*

inc (команда інкременту): приймач + 1 = приймач

sub (команда для віднімання одного числа з іншого): приймач - джерело =приймач

sbb (команда для віднімання одного числа з іншого з урахуванням прапора перенесення):приймач - джерело - CF = приймач

dec (команда декременту): приймач – 1 = приймач

neg (команда для зміни знака числа): -приймач = приймач

mul (команда множення чисел без знака): AL * джерело (8) = AX, AX * джерело (16) = DX: AX, EAX * джерело (32) = EDX: EAX

imul (команда множення чисел зі знаком): див. опис команди **mul** (прицьому операнди знакові)

div (команда поділу чисел без знака): AX/джерело (8) = AL:AH, AX / джерело (16) = AX: DX, EAX / джерело (32) = EAX: EDX

idiv (команда для виконання операції поділу чисел зі знаком): див.команду **div** (при цьому операнди знакові)

Ці команди називаються командами арифметичних операцій.

```
// розв'язати математичниq вираз
short Result, Remainder;
// (6+2*4)/(5-3)
_asm {
        // перший крок - перші дужки операція множення
        mov al, 2;
        mov bl, 4;
        mul bl;
        // операція додавання
        mov bx, 6;
        add ax, bx;
        // команда push - переміщує значення в стек
        push ax;// тимчасово сховаємо значення перших дужок в стек
        // другий крок - другі дужки операція віднімання
        mov ax, 5;
        mov bx, 2;
        sub ax, bx;
        // команда xchg - змінює значння двох регістрів або регістр<-> пам'ять
        xchg ax, bx;
        mov rez1, bx;
        // команда pop - отримує значення з стеку,
        //прицьому с стеку воно пропадає
        pop ax;
        xor dx,dx;
        div bx;
        mov rez1, ax;
        mov rez2, dx;
}
cout << "(6+2*4)/(5-2) = 14/3 = 4 end 2 remainder " << endl;
cout << "Result = " << Result << endl;
        cout << "Remainder = " << Remainder << endl;
```

Тут слід чітко розуміти, що при роботі з дійсними числами використовується окремий алгоритм а також спецпроцесор. Тому на даному етапі при виконанні операції ділення будемо враховувати тільки цілу частину.

Команда CMP віднімає вихідний операнд з операнда одержувача даних та, залежно від отриманого результату, встановлює прапори стану процесора. При цьому, на відміну від команди SUB значення операнда одержувача даних не змінюється.

стр одержувач, джерело

У команді CMP використовуються аналогічні команді та типи операндів.

Прапори. Команда CMP змінює стан наступних прапорів: CF (прапор перенесення), ZF (прапор нуля), SF (прапор знака), OF (прапор переповнення), AF (прапор службового перенесення), PF (прапор парності). Вони встановлюються залежно від значення, яке було отримано в результаті застосування команди SUB. Наприклад, як показано у таблиці, після виконання

команди CMP, станом прапорів нуля (ZF) і перенесення (CF) можна судити про величини порівнюваних між собою беззнакових операндів.

Значення операндів	**ZF**	**CF**
одержувач < джерело	0	1
одержувач > джерело	0	0
одержувач = джерело	1	0

Якщо порівнюються два операнди зі знаком, то, крім прапорів ZF і CF, потрібно враховувати ще й прапор знака (SF), як показано в таблиці.

Значення операндів	**Стан прапорів**
одержувач < джерело	SF ≠ OF та ZF = 0
одержувач > джерело	SF = OF та ZF = 0
одержувач = джерело	ZF = 1

Команда CMP дуже важлива, оскільки вона використовується практично у всіх основних умовних логічних конструкціях. Якщо після команди CMP помістити команду умовного переходу, то отримана конструкція мовою асемблера буде аналогічна оператору мови високого рівня.

Розглянемо фрагменти коду, де продемонстровано вплив команди CMP на прапори стану процесора. При порівнянні числа 5, що знаходиться в регістрі EAX, з числом 10, встановлюється прапор переносу CF , оскільки при відніманні числа 10 з 5 відбувається позика одиниці:

```
mov eax, 5
cmp eax, 10; CF = 1
```

При порівнянні вмісту регістрів eax і ecx, у яких містяться однакові числа 1000, встановлюється прапор нуля (ZF), оскільки в результаті віднімання цих чисел виходить нульове значення:

```
mov eax , 1000
mov ecx , 1000
cmp ecx , eax          ; ZF = 0 та CF = 0
```

Команда безумовного переходу має наступний синтаксис:

```
jmp <операнд>
```

Операнд вказує адресу переходу. Існує два способи вказівки цієї адреси, відповідно розрізняють прямий та непрямий переходи.

Прямий перехід. Якщо команді переходу вказується мітка команди, яку треба перейти, то перехід називається прямим .

```
jmp L
...
...
L: mov eax, x
```

Непрямий перехід. У разі непрямого переходу в команді переходу вказується не адреса переходу, а регістр або осередок пам'яті, де ця адреса знаходиться. Вміст вказаного регістра або осередку пам'яті розглядається як абсолютна адреса переходу. Непрямі переходи використовуються в тих випадках, коли адреса переходу стає відомою лише під час роботи програми.

```
jmp ebx
```

Команди умовного переходу. У системі команд процесора архітектури x86 не передбачено підтримки умовних логічних структур, притаманних мов високого рівня. Однак мовою асемблера за допомогою набору команд порівняння та умовного переходу ви можете реалізувати логічну структуру будь-якої складності. У мові високого рівня будь-який умовний оператор виконується у два етапи. Спочатку обчислюється значення умовного висловлювання, та був, залежно з його результату, виконуються ті чи інші дії. Проводячи аналогію з мовою асемблера, можна сказати, що спочатку виконуються такі команди, як **cmp** , **and** або **sub**, що впливають на прапори стану процесора Потім виконується команда умовного переходу, яка аналізує значення потрібних прапорів, і якщо вони встановлені, виконують перехід за вказаною адресою.

Що ж до команд умовного переходу, їх досить багато, але вони записуються однаково:

```
Jxx <мітка>
```

Усі команди умовного переходу можна розділити втричі групи.

До першої групи входять команди, які зазвичай ставляться після команди порівняння. У їх мнемокодах вказується результат порівняння, у якому треба робити перехід.

Менімокод	Назва	Умова переходу після команди CMP op 1 , op 2	Значення прапорів	Примітка
JE	Перехід якщо рівні	op1 = op2	ZF = 1	Для всіх чисел
JNE	Перехід якщо не рівні	op1 ≠ op2	ZF = 0	
JL/JNGE	Перехід якщо менше	op1 < op2	SF ≠ OF	Для чисел зі знаком
JLE/JNG	Перехід якщо менше чи рівні	op1 ≤ op2	SF ≠ OF або ZF = 1	
JG/JNLE	Перехід якщо більше	op1 > op2	SF = OF та ZF = 0	
JGE/JNL	Перехід якщо більше чи рівні	op1 ≥ op2	SF = OF	
JB/JNAE	Перехід якщо нижче	op1 < op2	CF = 1	Для чисел без знаку
JBE/JNA	Перехід якщо нижче або рівні	op1 ≤ op2	CF = 1 або ZF = 1	
JA/JNBE	Перехід якщо вище	op1 > op2	CF = 0 та ZF = 0	
JAE/JNB	Перехід якщо вище чи рівні	op1 ≥ op2	CF = 0	

До другої групи команд умовного переходу входять ті, які зазвичай ставляться після команд, відмінних від команди порівняння, і які реагують те чи інше значення будь-якого прапора.

Менімокод	Умова переходу	Менімокод	Умова переходу
JZ	ZF = 1	JNZ	ZF = 0
JS	SF = 1	JNS	SF = 0
JC	CF = 1	JNC	CF = 0
JO	OF = 1	JNO	OF = 0
JP	PF = 1	JNP	PF = 0

І, нарешті, до третьої групи входять дві команди умовного переходу, що перевіряють не прапори, а значення регістру **ecx** або **cx**:

jcxz <мітка>; Перехід, якщо значення регістра CX дорівнює 0
jecxz <мітка> ; Перехід, якщо значення регістра ECX дорівнює 0

Однак, ця команда виконується досить довго. Найвигідніше провести порівняння з нулем і використовувати звичайну команду умовного переходу.

Приклад реалізації конструкції if-then-else мовою асемблера. Наприклад перетворимо код C++:

```
if( a == b )
   a--;
else
   b = b +1;
```

в код мовою асемблера:

```
short a = 3, b = 3;
cout << "A = " << a << " B = " << b << endl;
_asm {
        mov ax, a;
        mov bx, b;
        cmp ax, bx;
        jne notEquals
                dec a;
        jmp endLine
        notEquals:
                inc b;
        endLine:
}
cout << "A = " << a << " B = " << b << endl;
//змінемо значення змінних a та b (щоб спрацювала гілка false)
//та для прикладу перепишемо математику для зміни значень змінних
a = 10, b = 5;
cout << "A = " << a << " B = " << b << endl;
_asm {
        mov ax, a;
        mov bx, b;
        cmp ax, bx;
        jne notEquals_2
                sub ax, 1;
                mov a, ax;
        jmp endLine_2
                notEquals_2 :
        add b,1;
endLine_2:
}
        cout << "A = " << a << " B = " << b << endl;
```

2. Структура звіту лабораторної роботи.

- Титульна сторінка.
- Відповіді на контрольні запитання.
- Програмний код розв’язку індивідуального завдання.
- Скріншоти роботи програмного додатку.
- Висновки.

3. Контрольні запитання

28. Що таке регістри процесора? Перерахуйте регістри загального призначення.
29. Що таке регістр прапорів і навіщо він призначений? Перерахуйте основні прапори регістру прапорів та поясніть їх призначення.

30. Які способи адресації у мові Асемблер ви знаєте?
31. Які дії виконує команда MOV? Опишіть формальне визначення цієї команди. Чи можна надіслати дані з пам'яті в пам'ять, використовуючи команду MOV?
32. Які дії виконує команда MOVZX? Опишіть формальне визначення цієї команди.
33. Які дії виконують команди LAHF та SAHF? Опишіть формальне визначення даних команд.
34. Навіщо використовується команда XCHG? Опишіть формальне визначення даної команди.

4. Завдання для індивідуального виконання

Для успішного виконання завдання необхідно розв’язати ряд задач, кожна з яких додає певну кількість балів до підсумкової оцінки:

1) Оголосити 3 змінні. За допомогою команд асемблера надіслати дані з 1-ї змінної в другу, з 2-ї в 3-ю.
2) Переслати дані в регістр AX окремими байтами (використовуючи мнемоніки AL і AH) і зберегти результат у 16-бітній змінній
3) Надіслати молодше слово регістру EAX в регістр EBX окремими байтами
4) Переслати дані регістру AX у дві 8-бітові змінні
5) Обміняти значення першої та третьої змінної
6) Обмінювати значення 2-х регістрів процесора з використанням оперативної пам'яті (змінних на C++)
7) Розв’язати математичний вираз обравши варіант згідно пори року коли ви народились (a- зима, b- весна, c – літо, d- осінь):
    a. x= ((12-4)/(3+5))*12
    b. x= ((3+21)*(14-2))/12
    c. x= (3*5+2)/(3-5*2)
    d. x= ((110/10)+(18-5))*2
8) Розв’язати приклад:

1. $y = y1 + y2;\ y1 = \begin{cases} a + x, & \text{если } x > a \\ 2a - x, & \text{если } x <= a \end{cases};\ y2 = \begin{cases} a * x, & \text{если } x > 10 \\ x, & \text{если } x <= 10 \end{cases}.$

2. $y = y1 - y2;\ y1 = \begin{cases} x - 2, & \text{если } x >= 2 \\ 8, & \text{если } x < 2 \end{cases};\ y2 = \begin{cases} 4, & \text{если } x = 0 \\ a - x, & \text{если } x <> 0 \end{cases}.$

3. $y = y1 * y2;\ y1 = \begin{cases} x - a, & \text{если } x > a \\ 5, & \text{если } x <= a \end{cases};\ y2 = \begin{cases} a, & \text{если a} > x \\ a * x, & \text{если a} <= x \end{cases}.$

4. $y = y1 + y2;\ y1 = \begin{cases} 2 - x, & \text{если } x < 2 \\ a + 3, & \text{если } x >= 2 \end{cases};\ y2 = \begin{cases} a - 1, & \text{если } x < a \\ a * x - 1, & \text{если } x >= a \end{cases}.$

5. $y = y1 - y2;\ y1 = \begin{cases} |x|, & \text{если } x < 0 \\ x - a, & \text{если } x >= 0 \end{cases};\ y2 = \begin{cases} a + x, & x \bmod 3 = 1 \\ 7, & \text{інакше} \end{cases}$

6. $y = y1 + y2;\ y1 = \begin{cases} x \bmod 4, & \text{если } x > a \\ a, & \text{если } x <= a \end{cases};\ y2 = \begin{cases} a * x, & \text{если } x / a > 3 \\ x, & \text{если } x / a <= 3 \end{cases}.$

7. $y = y1 + y2;\ y1 = \begin{cases} 4 - x, & \text{если } |x| < 3 \\ a + x, & \text{інакше} \end{cases}.\ y2 = \begin{cases} 2, & \text{если } x \text{ четное} \\ a + 2, & \text{інакше} \end{cases}$

8. $y = y1 + y2;\ y1 = \begin{cases} 4 * x, & \text{если } x <= 4 \\ x - a, & \text{если } x > 4 \end{cases};\ y2 = \begin{cases} 7, & \text{если } x \text{ нечетное} \\ x / 2 + a, & \text{інакше} \end{cases}$

9. $y = y1 * y2;\ y1 = \begin{cases} a * x, & \text{если } x \bmod 3 = 2 \\ 9, & \text{інакше} \end{cases};\ y2 = \begin{cases} a - x, & \text{если } a > x \\ a + 2, & \text{если } a <= x \end{cases}$

10. $y = y1 - y2;\ y1 = \begin{cases} a + |x|, & \text{если } x > a \\ a - 7, & \text{если } x <= a \end{cases};\ y2 = \begin{cases} a * 3, & \text{если } a > 3 \\ 11, & \text{если a} <= 3 \end{cases}$

11. $y = y1 \bmod y2;\ y1 = \begin{cases} 10 + x, & \text{если } x > 1 \\ |x| + a, & \text{если } x <= 1 \end{cases};\ y2 = \begin{cases} 2, & \text{если } x > 4 \\ x, & \text{если } x <= 4 \end{cases}$

12. $y = y1 / y2;\ y1 = \begin{cases} 15 + x, & \text{если } x > 7 \\ |a| + 9, & \text{если } x <= -7 \end{cases};\ y2 = \begin{cases} 3, & \text{если x} > 2 \\ |x| - 5, & \text{если } x <= 2 \end{cases}$

$y = y1 * y2;\ y1 = \begin{cases} 3 + x, & \text{если } x = a \\ a - x, & \text{если } x <> a \end{cases};\ y2 = \begin{cases} |a|, & \text{если } a < x \\ |a| - x, & \text{если a} >= x \end{cases}$

9) Результати виконання окремих кроків проілюструвати за допомогою виводу інформації на консоль

Лабораторна робота 6

Тема: Циклічні алгоритми алгоритми та операції над масивами з використанням вставок на мові Assembler.

Мета: Отримати практичні навики по реалізації циклічних алгоритмів та операції над масивами з використанням вставок на мові Assembler.

6. Теоретичні відомості

Як і у будь-якій мові програмування, у мові Assembler існує кілька способів організації циклічного повторення фрагмента програми. Кожен із способів має свої особливості, тому для ефективної реалізації конкретного завдання слід використовувати найбільш підходящий спосіб. Розглянемо особливості методів організації циклів.

Перший спосіб - організація циклу за допомогою команд умовного переходу та регістра ecx/cx .

Нагадаємо, що в архітектурі мікропроцесорів Intel регістр ecx/cx має певне функціональне призначення - виконує роль лічильника.

Якщо певну групу команд необхідно повторити певну кількість разів, цикл можна організувати так:

1) помістити в регістр ecx/cx кількість повторень;

2) першу команду тіла циклу відзначити міткою;

3) після виконання тіла циклу зменшити вміст регістру ecx/cx на 1;

4) виконати порівняння вмісту регістру ecx/cx з нулем;

5) у разі, якщо ecx/cx ≠ 0, здійснити перехід на мітку, інакше виконувати наступну за тілом циклу команду.

Схема реалізації виглядатиме таким чином:

```
    mov cx,N
CYCL:
    <тіло циклу>
    dec cx
    cmp cx,0
    jne CYCL
    ...
```

Увага. Необхідно звертати увагу на виконання «порожнього» циклу. Якщо раптово початкове значення в регістрі cx дорівнює нулю, то після зменшення на одиницю вмістом регістру стане число FFFFh (-1 в десятковій системі числення). В результаті цикл повториться ще 65535 разів (або 4294967295 разів при використанні регістра ecx). Тому слід завжди перевіряти вміст регістру ecx/cx для запобігання виконання «порожнього» циклу. Таку перевірку легко

організувати за допомогою команди умовного переходу jcxz/jecxz . Синтаксис команди:

jcxz <мітка переходу>

Або у разі необхідності використання розширеного регістру

jecxz <мітка переходу>

Переклад-розшифровка абревіатури назви команди Jump if cx is Zero — перехід, якщо cx дорівнює нулю. Зауваження. Команда jcxz/jecxz може адресувати лише короткі переходи (на –128 або +127 байт від наступної команди). Отже, схема організації циклу із запобіганням виконання «порожнього» циклу

```
    mov cx,N
    jcxz Exit
CYCL:
    <тіло циклу>
    dec cx
    cmp cx,0
    jne CYCL
Exit: . . .
```

Другий спосіб - організація циклу за допомогою команди безумовного переходу jmp і регістра ecx/cx .

Цикл можна організувати і так:

1) помістити в регістр ecx/cx кількість повторень;

2) здійснити перевірку на "порожній" цикл командою jecxz/jcxz <мітка переходу>

3) цю команду перевірки відзначити міткою початку циклу;

4) після виконання тіла циклу зменшити вміст регістру ecx/cx на 1;

5) здійснити безумовний перехід початку циклу.

Схема реалізації виглядатиме таким чином:

```
    mov cx,N
CYCL:
    jcxz Exit
    < тіло циклу >
    dec cx
    jmp CYCL
Exit: . . .
```

Звернемо увагу, що тут команда jcxz грає подвійну роль. По-перше, запобігає виконанню «порожнього» циклу, і, по-друге, відстежує закінчення циклу.

Третій спосіб - організація циклу за допомогою спеціальних команд loop.

Зауважимо, що у описаних перших двох способах організації циклу більшість операцій виконуються “вручну” (декремент регістру cx, порівняння cx із нулем, перехід початку циклу).

Враховуючи важливість циклів (практично жодна програма не обходиться без таких конструкцій), розробники системи команд мікропроцесора ввели спеціальні команди, що полегшують (що скорочують) програмування циклічних ділянок програм. Це команди:

loop <мітка переходу>
loope/loopz <мітка переходу>
loopne/loopnz <мітка переходу>

Зауважимо, ці команди також використовують регістр ecx/cx як параметр (лічильник) циклу.

Робота команди полягає у виконанні наступних дій:

1) зменшення значення регістра ecx/cx на 1;

2) порівняння регістра ecx/cx з нулем: якщо ecx/cx = 0, здійснюється вихід із циклу, тобто. управління передається наступну після loop команду; інакше - керування передається на мітку переходу (цикл повторюється).

Порядок організації циклу за допомогою команди loop:

1) у регістр cx помістити значення, що дорівнює кількості повторень;

2) встановити мітку на першу команду тіла циклу;

3) виконати команду loop <мітка переходу>.

Схема реалізації виглядає так:

```
    mov ecx,N
    jecxz Exit
CYCL:
    < тіло циклу >
    loop CYCL
Exit: . . .
```

Звернемо увагу також на те, що після виходу з циклу вміст регістра cx завжди дорівнює нулю.

Для прикладу знайдемо факторіал числа N. Код на ассемблері буде наступним. За основу використаємо варіант орагнізації циклу з командою loop:

```
short N = 5, factor = 1;
_asm {
		mov ax, factor;
		mov bx, ax;
		mov ecx, 5;// цикл буде мати 5 ітерацій
		jcxz endLine_3
	startLoop:
		mul bx;
		inc bx;
	loop startloop
	endLine_3:
		mov factor, ax;
}
	cout << "Factorial N=" << N << " => " << factor << endl;
```

Елементи масиву розташовуються в пам'яті комп'ютера послідовно. Доступ до елемента масиву зазвичай здійснюється операцією індексування, яку можна моделювати, знаючи початкову адресу масиву та розмір його елемента в байтах. Тоді адреса i-го (i=0,1,…) елемента одновимірного масиву дорівнює:

*початкова_адреса+(i*розмір_елемента).*

```
// масиви
int temp = 0;
char tempChar;
char mas[] = { 'a','1','b','M' };
int array1[] = { 123,65,67,100,102 };

_asm {
	mov ebx, 1;// ebx = зміщення від початку масиву в байтах
	add ebx, 1;
	mov dl, mas[ebx];
	// команда lea -  отримати адресу (фактично виходить вказівник)
	lea ebx, mas;
	// щоб взяти значення за адресом, пикористовуємо []
	mov dl, [ebx + 1];// У dl заноситься другий елемент масиву через адрес
	//mov dl, [mas+2];// У dl заноситься третій елемент масиву
	//mov dl, [mas]+1;// У dl заноситься другий елемент масиву
	//mov dl, 3[mas];// У dl заноситься четвертий елемент масиву
	//lea ebx, mas;
	//mov dl, byte ptr [ebx];
	mov ebx, 3;
	lea esi, mas;
	mov dl, [esi][ebx];
	mov tempChar, dl;

	mov ebx, 8;
	// передача змінна -> змінна на пряму неможлива, тому через регістр
	mov eax, array1[ebx];
	mov temp, eax;
}
cout << "tempCharp = " << tempChar << endl;
cout << "temp = " << temp << endl;
```

Розглянемо роботу з масивами мовою асемблера на прикладі: обчислити суми елементів одновимірного масиву. Здійснюється послідовний доступ до всіх елементів масиву, починаючи з 0-го. Використовується опосередковано-

реєстрова адресація: у регістрі ebx зберігається адреса поточного елемента масиву.

```
//сума всіх елементів масиву
int array2[] = { 4,53,45,34,5,34 };
int suma = 0;
int sizeArray2 = 6;
_asm {
                    mov eax, suma;
                    lea ebx, array2;
                    mov ecx, sizeArray2;
                    jecxz endLine_4
          startCycle:
                    add eax, [ebx];
                    add ebx, 4;
          loop startCycle
                    endLine_4:

          mov suma, eax;
}
cout << "Sum [4,53,45,34,5,34] = " << suma << endl;
```

2. Структура звіту лабораторної роботи.

- Титульна сторінка.
- Відповіді на контрольні запитання.
- Програмний код розв’язку індивідуального завдання.
- Скріншоти роботи програмного додатку.
- Висновки.

3. Контрольні запитання

35. Для яких цілей використовують цикли в програмуванні?
36. Що таке цикл із передумовою?
37. Чим відрізняється цикл while від циклу for?
38. Як організувати цикл while мовою асемблера?
39. Як у пам'яті елементи одномірного масиву?
40. Як звернеться до елемента масиву мовою C++?
41. Як отримати доступ у циклі до чергового елементу масиву мовою асемблера, використовуючи різні способи адресації?

4. Завдання для індивідуального виконання

Для успішного виконання завдання необхідно розв’язати ряд задач, кожна з яких додає певну кількість балів до підсумкової оцінки:

10) Даний масив цілих чисел: [12, 53, -5, 10, 11, -7, 20, 32, -12, 15]. Заповнити два інших масиви результатами цілочисельного ділення кожного з елементів на 5, та відповідно залишків від цілочисельного ділення.
11) Даний масив цілих чисел: [32, 10, 75, 41, -12, 52, 48, 61, 10, 11]. Заповнити два інших масиви квадратами та кубами цих чисел.
12) Дано два масиви цілих чисел: [52, 14, 17, -5, 23, 10, 41, -8, 9, 15] та [21, 17, 75, 41, 12, 52, 48, 11, 10, 11]. На основі цих масивів заповнити масиви сум і різниць їх елементів.
13) Дано масив цілих чисел: [52, 24, 17, -15, 33, 11, 48, 18, -9, 5]. Обчислити їх середнє арифметичне та заповнити інший масив, елементи якого рівні різниці вихідних елементів та отриманого середнього.
14) Дано масив цілих чисел: [17, -5, 23, -12, 24, 48, 61, 73, -9, 15]. Обчислити їх мінімальне значення та заповнити інший масив, елементи якого рівні різниці вихідних елементів та отриманого мінімального.
15) Дано масив цілих чисел: [17, -5, 3, -12, 24, 78, 61, 73, -9, 13]. Обчислити їх максимальне значення та заповнити інший масив, елементи якого рівні різниці отриманого максимального та вихідних елементів.
16) Дано два масиви цілих чисел: [41, 11, 19, -10, 23, -13, 41, 18, 9, 17] і [23, 45, 35, 41,12, -53, 48, 17, 10, 15]. На основі цих масивів заповнити масив, елементи якого будуть рівні різниці відповідних елементів у другому степені.
17) Даний масив цілих чисел: [12, 13, -15, 10, 11, -17, 25, 32, 14, 15]. Заповнити два інших масиви результатами цілочисельного ділення кожного з елементів на 3, та відповідно залишків від цілочисельного ділення.
18) Дано масив цілих чисел: [28, 64, 35, 20, 43, -15, 75, 44, -20, 15]. Знайти різницю між максимальним та мінімальним елементами масиву.
19) Даний масив цілих чисел: [43, -25, 14, 16, -25, 78, 69, 96, -18, 23]. Визначити скільки в ньому є парних чисел.

ЛІТЕРАТУРА

Список рекомендованої літератури:

1. Лав Р. Linux. Системное программирование. - СПб.: Питер, 2018. - 416 с.: ил. - (Серия«Бестселлеры O'Reilly»).
2. Э.Таненбаум: Операционные системы, Разработка и реализация, Питер,2016. - 762с.
3. Бек, Л. Введение в системное программирование / Л. Бек. - М.: Мир, 2016. - 448 с.
4. Кетков, Ю.Л. Введение в системное программирование на языке ассемблера ЕС ЭВМ / Ю.Л. Кетков, В.С. Максимов, А.Н. Рябов. - М.: Наука, 2018. - 264 с.
5. Кнут, Д.Э. Искусство программирования (Том 1. Основные алгоритмы) / Д.Э. Кнут. - М., 2017. - 882 с.
6. Кнут, Д.Э. Искусство программирования (Том 2. Получисленные алгоритмы) / Д.Э. Кнут. - М., 2017. - 858 с.
7. Лав, Р. Linux. Системное программирование / Р. Лав. - М.: Питер, 2016. - 456 с.
8. Макаров, А. В. Common Intermediate Language и системное программирование в Microsoft. NET / А.В. Макаров, С.Ю. Скоробогатов, А.М. Чеповский. - М.: Интернет-университет информационных технологий, Бином. Лаборатория знаний, 2018. - 328 с.
9. Макаров, А.В. Common Intermediate Language и системное программирование в Microsoft. NET: моногр. / А.В. Макаров. - М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. - 573 с.
10. Хювёнен, Э. Мир Лиспа. Том 2. Методы и системы программирования / Э. Хювёнен, И. Септянен. - М, 2019. - 742 с.