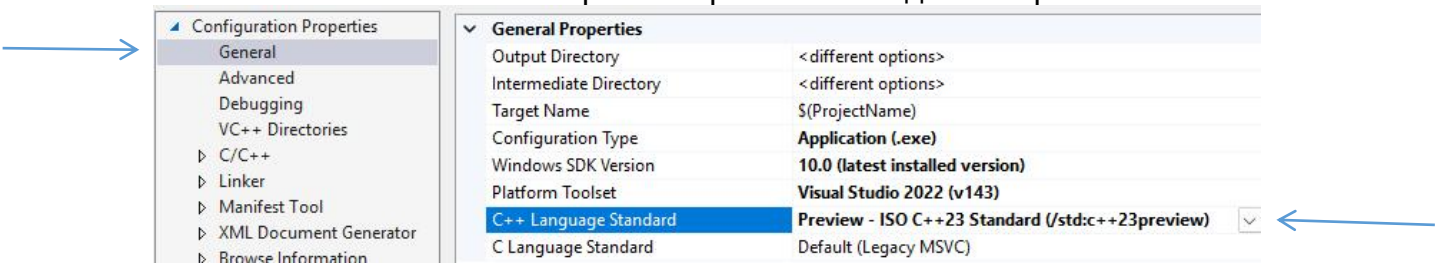
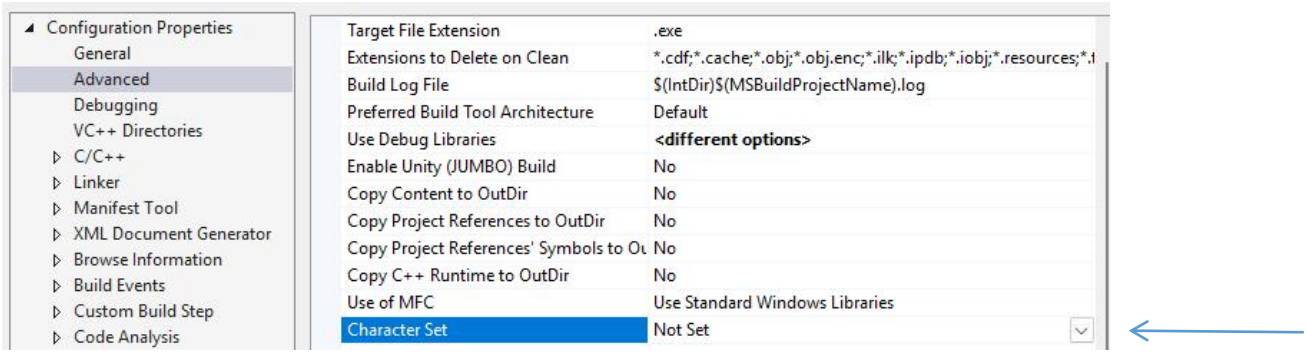
Руководство для создания простого windows окна

Подключаем #include "windows.h"

Создаем класс в отдельном заголовочном файле .h, будем его использовать для создания окна

Изначально поставим в настройках проекта последнюю версию языка





Приватная часть класса: private:
В него мы указываем название класса окна:

```
const char* NameClass = "Window";       - Название класса окна
RECT rc;                                - Контейнер
HINSTANCE hIns;                         - Экземпляр окна
HWND hWnd;                              - Дескриптор окна
```

Публичная часть класса: public:
Контруктор класса будет принимать (высоту, ширину окна, и название окна)

```
Window(int Width, int Height, const char* NameWind){реализация};
```

Внутри будет реализовано создание окна
Сначало мы заполним контейнер для размеров окна:

```
rc = { 0,0,Width,Height };
```

Рассчитаем оптимальный размер для окна при помощи функции AdjustWindowRect()

```
AdjustWindowRect(
&rc,                    - контейнер RECT
WS_CAPTION |            - стиль окна чтобы окно имело титульный бар
WS_MINIMIZEBOX |        - окно имеет кнопки
WS_SYSMENU,             - окно имеет меню (нужно указывать из-за стиля
                                                    WS_MINIMIZEBOX)
FALSE                   - указывает если меню у окна
);
```

Прочие стили можно найти здесь: https://learn.microsoft.com/en-us/windows/win32/winmsg/window-styles

Регистрация окна в системе:
Использвуем дискриптор WNDCLASSEX

```
WNDCLASSEX wc = { 0 };
wc.cbSize = sizeof(wc);                 - размер дискриптора в байтах
wc.lpszClassName = NameClass;           - имя нашего класса
wc.hInstance = hIns;                    - экзепляр окна
wc.lpfnWndProc = &WindowProc;           - это нужно для приемки сообщений (приемку сообщений мы созданим чуть позже)
```

Теперь используем функцию регистрации окна:

```
auto NameClassId = RegisterClassEx(&wc);
```

Создадим нашего окно при помощи функции CreateWindowEx()

```
hWnd = CreateWindowEx(
NULL,                                   - указываем дополнительные стили окна
MAKEINTATOM(NameClassId),               - регистрационный номер окна
NameWind,                               - имя окна
WS_CAPTION | WS_MINIMIZEBOX | WS_SYSMENU,   - базовые стили
CW_USEDEFAULT,                          - положение окна по X
CW_USEDEFAULT,                          - положение окна по Y
rc.right - rc.left,                     - ширина окна
rc.bottom - rc.top,                     - высота окна
NULL,                                   - родительское окно
NULL,                                   - десприптор для меню
hIns,                                   - экземпляр окна
NULL                                    - дополнительные данные для создания окна если мы получаем сообщение о его создании
);
```

Отобразим окно при помощи функции: ShowWindow()

```
ShowWindow(hWnd, SW_SHOW);
```

Мы закончили описывать конструктор теперь опишем деструктор класса:
В деструкторе мы просто будем удалять окно при завершении программы

```
~Window()
{
	DestroyWindow(hWnd);
};
```

Так же добавим еще простую функцию для доступа к нашему дескриптору окна для других программ:

```
HWND GetHWND()
{
	return hWnd;
}
```

Мы создали класс для создания окна после класса мы можем сразу создать окно чтобы оно было доступно сразу

```
Наш класс создания окна
{
...
}win(window.width, window.height, "GameFrog");
```

Чтобы получить размер вашего экрана можно использовать функции и заложить их в структуру для удобства использования например структура window

```
int width = GetSystemMetrics(SM_CXSCREEN);
int height = GetSystemMetrics(SM_CYSCREEN);
```

Остается решить проблему с получение сообщений.
Над классом созданием окна создаем функцию для примемки сообщений:

```
static LRESULT CALLBACK WindowProc(
HWND hWnd,                                  - дескриптор окна
UINT msg,                                   - код сообщения
WPARAM wParam,                              - параметры типа w
LPARAM lParam)                              - параметры типа l
```

Реализация нашей функции будет выглядеть так:

```
switch (msg)                                    -программа получает сообщение
{
case WM_CLOSE:                                   - код сообщение закрыть
                                                                приложение
        PostQuitMessage(0);                              -  передать код
                                                                программе 0
        break;
default:
        return DefWindowProc(hWnd, msg, wParam, lParam);        -базовое
                                                                сообщение
}
```

Созданием окна и приемку сообщений мы выполнили, но нам нужен вход в программу для этого мы создадим простой класс, который будет использоваться для отправки команд (сообщений), будет загружать входные данные при первой загрузке приложения, и будет являться циклом программы, пока она не будет выключена.

В классе будет базовый конструктор который не будет ничего принимать и ничего делать.

Будет реализована функция запуска программы в public: void FrameGo()

Перед этим в приватной части добавим функции для работы приложения:

```
private:
void UpdateApp(MSG* msg);
void Render();
void Init();
```

Напишем реализацию функции FrameGo()

```
MSG msg;                    - сообщение
BOOL gbool = true;          - логическая перменная для работы программы
Init();                     - подгрузка начальных данных программы
```

Теперь опишем основной цикл работы нашей программы:

```
while (gbool)
{
	while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) - отправляет
	                                                    сообщения в
	                                                        потоке
	{
		UpdateApp(&msg); - обработка различных функциональных
		                                                    сообщений
		if (msg.message == WM_QUIT)
		{
			gbool = false;
			break;
		}
		TranslateMessage(&msg);          - перевод сообщения в
		                                                        код
		DispatchMessage(&msg);      - отправка сообщения в окно
	}
	Render();                                - обновление приложения
}
```

Реализацию функций UpdateApp(MSG* msg);, Render();, Init();
Мы описываем отдельно

Пример для Init()

```
void AppGame::Init()
{
	LoadSVGDataMap(MAPS"LVLDemoDay0");
	LoadSVGDataMap(MAPS"LVLDemoDay1");

	MapSizeW = VLocation[0].GetPosition()->Width;
	MapSizeH = VLocation[0].GetPosition()->Height;
}
```

Временная реализация постоянного обновления для Render()

```
void AppGame::Render()
{
	Sleep(16);
}
```

Реализация для UpdateApp(MSG* msg)

```
void AppGame::UpdateApp(MSG* msg)
{
	if (GetAsyncKeyState(VK_ESCAPE))
	{
		msg->message = WM_QUIT;
	}
}
```

Финальными действиями будет запустить окно в main.cpp файле

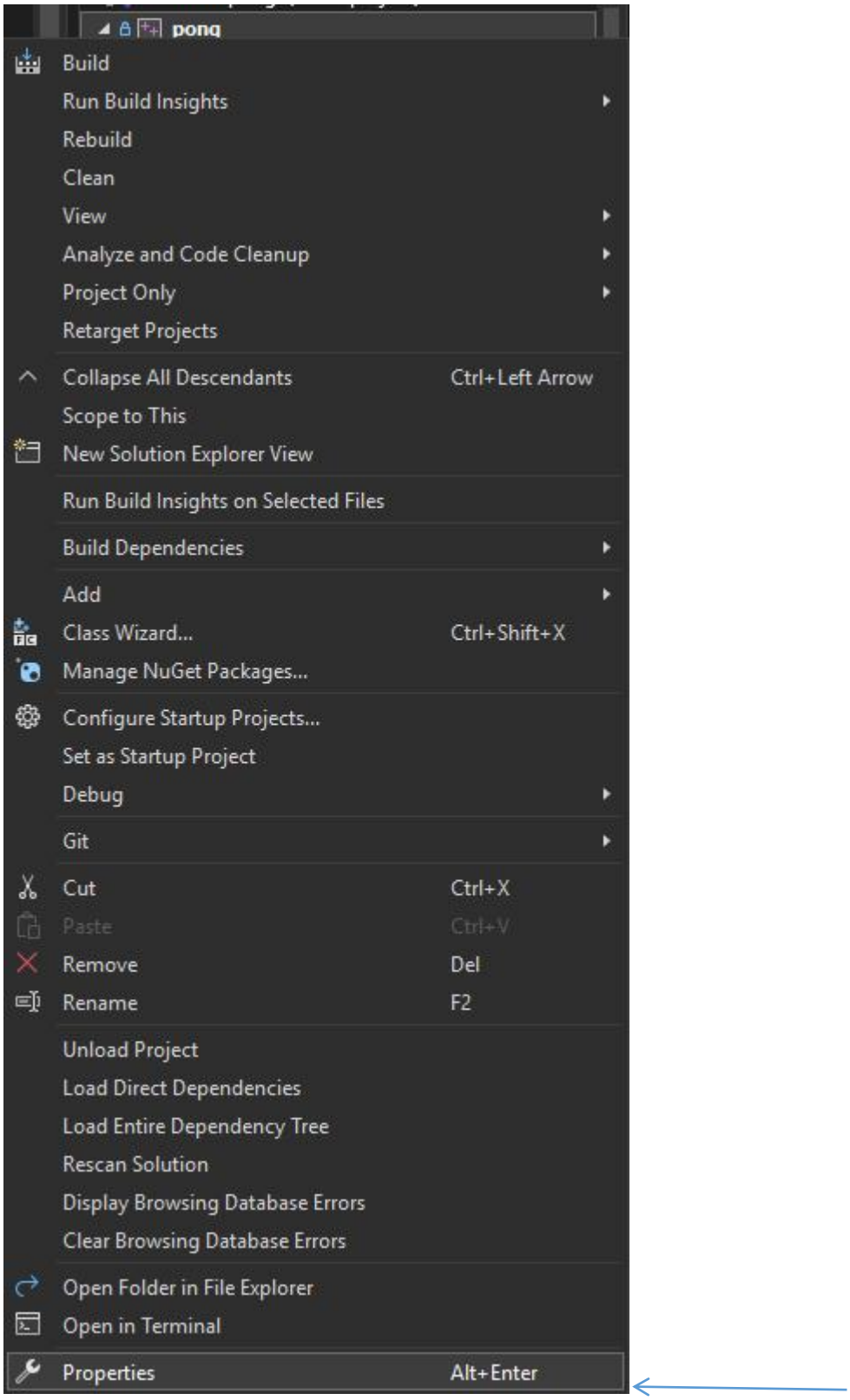
Для этого используем эту конструкцию

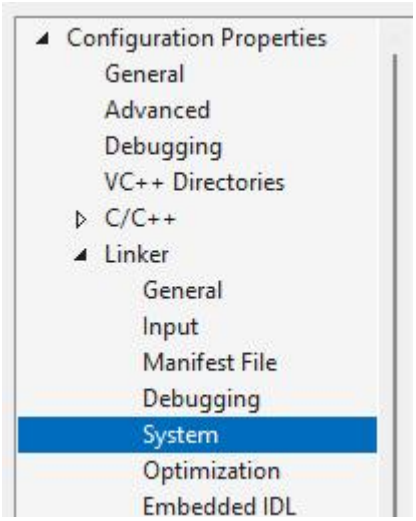
```
int CALLBACK WinMain(
	HINSTANCE hInstance,
	HINSTANCE hPrevInstance,
	LPSTR lpCmdLine,
	int nShowCmd)
{
	App.FrameGo();
	return 0;
};
```

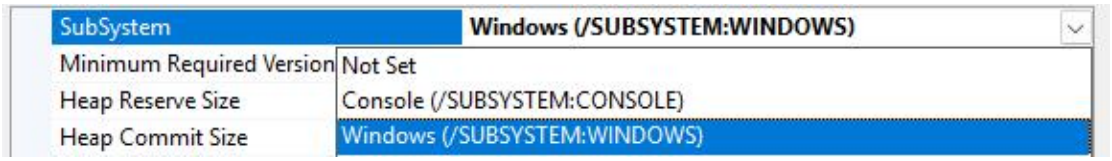
По сути она может работать и в таком виде

```
int main()
{
App.FrameGo();
return 0;
};
```

Это зависит от настройки проекта которые вы поставите
Настройки описаны ниже:







Для Int main подойдет консоль

а для winapi winows

Репозиторий с проектом:
https://github.com/wordlol/pract3d/tree/window_app