Ein neues WinUI 3-Projekt in **C++** (über C++/WinRT) ist eine hervorragende Wahl für maximale Performance und ressourceneffiziente Windows-Apps. Da C++ bei WinUI 3 etwas mehr Ersteinrichtung und Code-Struktur („Boilerplate“) erfordert als C#, hilft Ihnen der folgende Leitfaden beim optimalen Start. [1]

Voraussetzungen für den Start

Um das Projekt aufzusetzen, benötigen Sie **Visual Studio** (empfohlen: aktuelle Version). Stellen Sie im Visual Studio Installer sicher, dass folgende Workloads und Komponenten aktiv sind: [2]

- **Entwicklung für die universelle Windows-Plattform**
- **Desktopentwicklung mit C++**
- **Windows App SDK C++-Vorlagen** (oft automatisch enthalten oder als Erweiterung im Marktplatz unter "Windows App SDK Extension" zu finden) [3, 4]

Das erste Projekt erstellen

1. Öffnen Sie Visual Studio und wählen Sie **Neues Projekt erstellen**.
2. Filtern Sie die Sprache nach **C++** und die Plattform nach **Windows**.
3. Suchen Sie nach **Blank App, Packaged (WinUI 3 in Desktop)**. Achten Sie genau darauf, die **C++-Version** zu wählen (nicht C#).
4. Vergeben Sie einen Namen und erstellen Sie das Projekt. [5, 6, 7, 8]

Die Projektstruktur verstehen

Ein frisches C++ WinUI 3-Projekt unterscheidet sich strukturell von klassischen C++-Konsolenanwendungen:

- `MainWindow.xaml`: Hier definieren Sie die Benutzeroberfläche deklarativ mit XAML (z. B. Schaltflächen, Layouts, Textfelder).
- `MainWindow.idl`: Die *Interface Definition Language*-Datei. Sie ist der Schlüssel in C++/WinRT. Jede Methode und jede Eigenschaft, die Ihr XAML-Code aufrufen soll (z. B. Klick-Events), muss hier zuerst deklariert werden.
- `MainWindow.xaml.h` **und** `.cpp`: Hier schreiben Sie die eigentliche Logik für Ihr Hauptfenster in nativem C++. [9]

Wichtiger Workflow-Tipp (IDL-Kompilierung)

Der häufigste Stolperstein für C++ WinUI-Einsteiger ist das Hinzufügen von neuen Events (z. B. ein Button-Klick):

1. **In XAML:** Sie fügen ein Element hinzu (z. B. `<Button Click="myButton_Click"/>`).
2. **In IDL:** Sie müssen die Methode in der `.idl`-Datei eintragen.
3. **Projekt kompilieren:** Drücken Sie `Strg + Umschalt + B`. Dadurch generiert das Windows App SDK im Hintergrund die passenden C++-Header-Dateien (zu finden im Ordner `Generated Files` im Projektverzeichnis).
4. **In CPP einfügen:** Kopieren Sie die generierte Methodensignatur aus den `Generated Files` in Ihre `MainWindow.xaml.h` und `MainWindow.xaml.cpp` und füllen Sie sie mit Logik.

Um Ihnen zielgerichtet weiterzuhelfen, lassen Sie mich wissen:

- Welche **Art von Anwendung** planen Sie (z. B. Tool, Grafik/Audio, Systemsoftware)?
- Benötigen Sie Unterstützung beim **ersten XAML-Layout** oder der **IDL-Syntax**?

Für die Entwicklung mit **WinUI 3** stellt Microsoft offiziell zwei primäre Programmiersprachen zur Auswahl: **C# (.NET)** und **C++**. Für die Gestaltung der Benutzeroberfläche wird in beiden Fällen standardmäßig **XAML** als deklarative Markup-Sprache genutzt. [1, 2, 3]

Hier ist eine Übersicht, welche Sprache für Ihr Projekt am besten geeignet ist:

1. C# (Die Empfehlung für die meisten Entwickler)

Für den Großteil aller WinUI 3-Projekte ist **C# in Kombination mit .NET** die am häufigsten gewählte und am besten unterstützte Option. [2, 4]

- **Vorteile:** Deutlich schnellere Entwicklungszeit, automatische Speicherverwaltung (Garbage Collection), exzellenter Tool-Support und eine riesige Auswahl an Bibliotheken via NuGet.
- **Wann wählen?** Für klassische Desktop-Anwendungen, Business-Software, datengetriebene Programme und wenn Sie eine moderne, produktive Entwicklungsumgebung schätzen. [3, 5]

2. C++ (Für maximale Performance)

Wenn Ihre App systemnah agieren muss oder extrem ressourcenintensiv ist, ist **C++ (über C++/WinRT)** die richtige Wahl. [6]

- **Vorteile:** Maximale Ausführungsgeschwindigkeit, minimale Dateigrößen der Anwendung, kein Overhead durch eine Runtime (.NET) und direkter Zugriff auf hardwarenahe Windows-APIs. [3, 6]
- **Wann wählen?** Für hardwarehungrige Apps wie 3D-Grafikprogramme, Audio-/Videobearbeitung, Spiele-Engines oder wenn Sie bestehenden C++-Code migrieren und wiederverwenden möchten.

Direktvergleich für die Entscheidungsfindung

Kriterium	C# / .NET	C++ (C++/WinRT)
Lernkurve	Flach, sehr einsteigerfreundlich	Steil, erfordert tiefes Verständnis
Entwicklungsgeschwindigkeit	Sehr hoch	Eher langsam (mehr Boilerplate-Code)
Performance	Sehr gut (für 95 % aller Apps absolut ausreichend)	Exzellent (native Hardwarenähe)
Dokumentation & Community	Sehr groß, viele Vorlagen und Code-Beispiele	Kleiner, oft komplexere Beispiele

---

Entwicklungsumgebung

Egal ob Sie sich für C# oder C++ entscheiden, das empfohlene Werkzeug von Microsoft ist **Visual Studio** mit der installierten Workload **„WinUI-Anwendungsentwicklung“** (Windows App SDK). [4]

Möchten Sie eine **bestehende Anwendung** (z. B. aus WPF oder C++) auf WinUI 3 migrieren, oder starten Sie ein **völlig neues Projekt**? Wenn Sie mir Details verraten, helfe ich Ihnen gerne bei der Architekturentscheidung. [7]

IDL-Syntax, warum diese COM Schnittstelle,das macht es doch sehr schwierig?

Es ist absolut verständlich, dass die IDL-Syntax und das zugrunde liegende COM-Modell (Component Object Model) auf den ersten Blick abschreckend und unnötig kompliziert wirken. Gerade wenn man modernes C++ gewohnt ist, fühlt sich IDL wie ein Rückschritt in die 90er-Jahre an.

Hinter dieser Design-Entscheidung von Microsoft steckt jedoch ein gewaltiger technischer Grund.

Warum nutzt WinUI 3 überhaupt COM?

WinUI 3 ist kein reines C++-Framework. Es ist Teil des **Windows App SDK**, dessen Kernkomponenten in nativem C++ geschrieben sind, das aber von **jeder** Sprache (C#, C++, Rust, Python) genutzt werden soll.

- **Das Problem von Standard-C++:** C++ hat kein standardisiertes "Binary Interface" (ABI). Das bedeutet: Wenn Sie eine C++-Klasse mit Visual Studio 2022 kompilieren, kann ein C#-Programm oder ein mit einem anderen Compiler geschriebenes Programm diese Klasse nicht direkt im Speicher aufrufen.
- **Die Lösung (COM/WinRT):** COM ist die einzige Technologie in Windows, die ein sprachunabhängiges, stabiles Binärformat garantiert. Die IDL-Datei ist das Bindeglied. Sie beschreibt die Schnittstelle so neutral, dass die WinUI-Engine (C++) und Ihre Anwendungslogik (C++) perfekt miteinander kommunizieren können, ohne dass sich Compiler-Details in den Weg stellen.

Warum macht es das für Sie so schwierig?

Das Problem ist die **Brücke zwischen XAML und C++**.
WinUI 3 nutzt das XAML-System, um die Oberfläche zur Laufzeit dynamisch aufzubauen. Damit das XAML-System (das Steuerelemente reflektieren und finden muss) Ihre C++-Klassen (wie `MainWindow`) versteht, müssen diese Klassen als COM-Objekte exportiert werden.

Jedes Mal, wenn Sie ein Event (wie einen Button-Klick) oder eine Datenbindung (`{x:Bind}`) nutzen, greift das XAML-System über diese COM-Schnittstelle darauf zu. Ohne die `.idl`-Datei wüsste das XAML-System schlicht nicht, dass Ihre C++-Klasse diese Methode besitzt.

Der Trost: C++/WinRT nimmt Ihnen 90 % der COM-Arbeit ab

Früher (mit altem COM oder WRL) musste man Referenzzähler manuell verwalten (`AddRef` / `Release`) und kryptische Pointer nutzen. Heute nutzt WinUI 3 **C++/WinRT**. Das ist ein moderner C++17/C++20-Wrapper.

- Sie nutzen moderne Smart Pointer (`winrt::com_ptr`).
- Die Speicherverwaltung erfolgt automatisch über RAII (wie `std::shared_ptr`).
- Sie schreiben in Ihrer `.cpp`-Datei ganz normales C++.

Die `.idl`-Datei dient wirklich **nur noch als Deklaration** für die Außenwelt (das XAML-System).

Die IDL-Syntax im Schnelldurchlauf (Es ist weniger als man denkt)

In 95 % der Fälle müssen Sie in der `.idl`-Datei nur neue Methoden oder Eigenschaften eintragen. Die Syntax ist sehr nah an C#:

```
// MainWindow.idl
namespace MeinProjekt
{
```

```
    [default_interface]
    runtimeclass MainWindow : Microsoft.UI.Xaml.Window
    {
        MainWindow();

        // Ein einfacher Button-Klick:
        void MyButton_Click(IInspectable sender,
Microsoft.UI.Xaml.RoutedEventArgs e);

        // Eine Eigenschaft (Property) für Datenbindung:
        String CustomTitle{ get; set; };
    }
}
```

- **`IInspectable`** ist das COM-Äquivalent zu `Object` (die Basisklasse für alles).
- **`runtimeclass`** sagt dem Compiler: "Mach hieraus eine COM-Klasse, die das XAML-System sehen kann."

Um den Einstieg zu erleichtern, kann ich Ihnen zeigen, wie Sie **ein konkretes Event** (z. B. Textänderung in einem Eingabefeld) Schritt für Schritt von der IDL bis zur CPP-Datei umsetzen. Wäre ein solches **Code-Beispiel** hilfreich für Sie?

IDL+ Datenbank Anwendung in C++

Für eine Datenbankanwendung in C++ mit WinUI 3 benötigen Sie die IDL-Syntax vor allem für zwei Dinge: **Methoden** (z. B. zum Laden von Daten bei einem Button-Klick) und **Eigenschaften (Properties)**, damit die UI die Daten aus der Datenbank über Datenbindung (`{x:Bind}`) anzeigen kann.

Hier ist ein durchgehendes, praxisnahes Beispiel für eine einfache Adressbuch-Datenbankstruktur (z. B. mit SQLite).

---

1. Das Datenmodell (IDL, Header, CPP)

Damit das XAML-System die geladenen Datenbankzeilen lesen kann, müssen wir einen Datentyp (z. B. `Kunde`) in der IDL definieren.

`Kunde.idl` (Neu anlegen)

Rechtsklick auf das Projekt -> Hinzufügen -> Neues Element -> Midl-Datei (`.idl`).

```
namespace MeinProjekt
{
    [default_interface]
    runtimeclass Kunde
    {
        Kunde(Int32 id, String name, String email);
```

```
        Int32 Id{ get; };
        String Name{ get; set; };
        String Email{ get; set; };
    }
}
```

Nach dem Speichern einmal das Projekt kompilieren (`Strg + Umschalt + B`), damit Visual Studio die C++-Klassenvorlage generiert.

`Kunde.h` (Neu anlegen)

Erstellen Sie die passende C++-Header-Datei. Sie erbt von der generierten WinRT-Basisklasse.

```
#pragma once
#include "Kunde.g.h" // Wichtig: Wird automatisch generiert

namespace winrt::MeinProjekt::implementation
{
    struct Kunde : KundeT<Kunde>
    {
        Kunde() = default;
        Kunde(int32_t id, hstring const& name, hstring const& email);

        int32_t Id();
        hstring Name();
        void Name(hstring const& value);
        hstring Email();
        void Email(hstring const& value);

    private:
        int32_t m_id{ 0 };
        hstring m_name;
        hstring m_email;
    };
}

namespace winrt::MeinProjekt::factory_implementation
{
    struct Kunde : KundeT<Kunde, implementation::Kunde> {};
}
```

`Kunde.cpp` (Neu anlegen)

Die Implementierung der Get-/Set-Methoden in C++:

```
#include "pch.h"
#include "Kunde.h"
#include "Kunde.g.cpp"

namespace winrt::MeinProjekt::implementation
{
    Kunde::Kunde(int32_t id, hstring const& name, hstring const& email) :
        m_id(id), m_name(name), m_email(email) {}

    int32_t Kunde::Id() { return m_id; }
    hstring Kunde::Name() { return m_name; }
    void Kunde::Name(hstring const& value) { m_name = value; }
    hstring Kunde::Email() { return m_email; }
    void Kunde::Email(hstring const& value) { m_email = value; }
```

```
}
```

---

2. Das Hauptfenster erweitern (IDL & C++)

Jetzt müssen wir dem Hauptfenster beibringen, dass es eine Liste von Kunden aus der Datenbank laden und halten kann. WinUI nutzt dafür `IObservableVector`, damit die UI automatisch merkt, wenn Zeilen hinzukommen.

`MainWindow.idl` (Anpassen)

```
import "Kunde.idl"; // Importiert unser Datenmodell

namespace MeinProjekt
{
    [default_interface]
    runtimeclass MainWindow : Microsoft.UI.Xaml.Window
    {
        MainWindow();

        // Die Liste, an die das XAML-Element (ListView) gebunden wird
        Windows.Foundation.Collections.IObservableVector<Kunde>
KundenListe{ get; };

        // Methode, die aufgerufen wird, wenn man auf "Laden" klickt
        void DatenbankLaden_Click(IInspectable sender,
Microsoft.UI.Xaml.RoutedEventArgs e);
    }
}
```

Wieder einmal kompilieren (`Strg + Umschalt + B`).

`MainWindow.xaml.h` (Anpassen)

Fügen Sie das Listen-Objekt und die Klick-Methode hinzu.

```
#pragma once
#include "MainWindow.g.h"
#include "Kunde.h" // Wichtig

namespace winrt::MeinProjekt::implementation
{
    struct MainWindow : MainWindowT<MainWindow>
    {
        MainWindow();

        // Getter für die IDL-Property

winrt::Windows::Foundation::Collections::IObservableVector<MeinProjekt::Kun
de> KundenListe();

        // Klick-Event
        void DatenbankLaden_Click(winrt::Windows::Foundation::IInspectable
const& sender, winrt::Microsoft::UI::Xaml::RoutedEventArgs const& e);

    private:
        // Hier liegen die echten Daten im Speicher
```

```
winrt::Windows::Foundation::Collections::IObservableVector<MeinProjekt::Kunde> m_kundenListe;
    };
}
// ... factory_implementation bleibt unverändert ...
```

`MainWindow.xaml.cpp` (Anpassen)

Hier simulieren wir den Datenbank-Select. Für echte Datenbanken würden Sie hier z. B. `sqlite3_prepare_v2` nutzen.

```
#include "pch.h"
#include "MainWindow.xaml.h"
#if __has_include("MainWindow.g.cpp")
#include "MainWindow.g.cpp"
#endif

using namespace winrt;
using namespace Microsoft::UI::Xaml;

namespace winrt::MeinProjekt::implementation
{
    MainWindow::MainWindow()
    {
        // Vektor initialisieren
        m_kundenListe = winrt::single_threaded_observable_vector<MeinProjekt::Kunde>();
        InitializeComponent();
    }


winrt::Windows::Foundation::Collections::IObservableVector<MeinProjekt::Kunde> MainWindow::KundenListe()
    {
        return m_kundenListe;
    }

    void MainWindow::DatenbankLaden_Click(IInspectable const&, RoutedEventArgs const&)
    {
        m_kundenListe.Clear();

        // HIER: Ihre SQL-Abfrageschleife (z.B. sqlite3_step)
        // Wir erstellen exemplarisch zwei COM-Objekte aus der "Datenbank"
        auto kunde1 = winrt::make<winrt::MeinProjekt::implementation::Kunde>(1, L"Max Mustermann", L"max@test.de");
        auto kunde2 = winrt::make<winrt::MeinProjekt::implementation::Kunde>(2, L"Erika Mustermann", L"erika@test.de");

        // UI aktualisiert sich durch Append() dank IObservableVector sofort automatisch
        m_kundenListe.Append(kunde1);
        m_kundenListe.Append(kunde2);
    }
}
```

---

3. Die Benutzeroberfläche (XAML)

Dank der IDL-Definitionen weiß das XAML-System nun exakt, was `KundenListe`, `Name` und `Email` bedeuten.

`MainWindow.xaml`

```
<Window
    x:Class="MeinProjekt.MainWindow"
    xmlns="http://microsoft.com"
    xmlns:x="http://microsoft.com"
    xmlns:local="using:MeinProjekt">

    <StackPanel Padding="20" Spacing="10">
        <Button Content="Daten aus DB laden" Click="DatenbankLaden_Click"/>

        <!-- Steuerelement zur Anzeige der Datenbankzeilen -->
        <ListView ItemsSource="{x:Bind KundenListe}">
            <ListView.ItemTemplate>
                <DataTemplate x:DataType="local:Kunde">
                    <StackPanel Orientation="Horizontal" Spacing="20">
                        <TextBlock Text="{x:Bind Id}" FontWeight="Bold"/>
                        <TextBlock Text="{x:Bind Name}"/>
                        <TextBlock Text="{x:Bind Email}"
Foreground="Gray"/>
                    </StackPanel>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </StackPanel>
</Window>
```

Zusammenfassung des Workflows

Obwohl die IDL-Schnittstelle extra Code erzwingt, sorgt sie hier für eine saubere Trennung: Die UI (`ListView`) greift über standardisierte COM-Wege auf Ihre `Kunde`-Objekte zu, während Sie im Hintergrund in der `.cpp`-Datei mit performantem, nativem C++ Ihre Datenbank-Abfragen abarbeiten können.

Möchten Sie im nächsten Schritt wissen, wie Sie eine **konkrete C++-Datenbankbibliothek** (wie SQLite oder libpq für PostgreSQL) in dieses WinUI 3-Projekt einbinden?

Als Datenbank, Vorschlag, Access, kein SQLite,

Um eine **Microsoft Access-Datenbank (`.accdb` oder `.mdb`)** nativ in einer modernen C++ WinUI 3-Anwendung anzusprechen, nutzt man unter Windows am stabilsten die klassische **ODBC-Schnittstelle (Open Database Connectivity)**. [1]

Da WinUI 3-Apps standardmäßig als moderne 64-Bit-Anwendungen (x64) kompiliert werden, ist ein technischer Fallstrick besonders wichtig.

Der wichtigste Fallstrick: Die Bit-Architektur (32-Bit vs. 64-Bit)

Damit Ihr C++-Code die Access-Datenbank öffnen kann, muss die Bit-Version Ihrer App exakt mit der Bit-Version der installierten Microsoft Office/Access-Engine auf dem PC übereinstimmen: [2]

- Ist auf dem PC ein **64-Bit Office** installiert, muss Ihre WinUI 3 App zwingend auf **x64** laufen.
- Ist ein **32-Bit Office** installiert, müssen Sie Ihr WinUI 3-Projekt in Visual Studio auf **x86** umstellen.
- Sollte auf dem Ziel-PC gar kein Access/Office installiert sein, muss die kostenlose Microsoft Access Database Engine Runtime (ACE) nachinstalliert werden. [2, 3]

Der C++ ODBC-Code für Access

In C++ nutzt man die Windows-eigenen Header `<sqlext.h>` und `<windows.h>`, um via ODBC SQL-Befehle an Access zu senden. [1, 4]

Hier ist das konkrete Beispiel, wie Sie die simulierte Methode `DatenbankLaden_Click` aus dem vorherigen IDL-Schritt durch eine **echte Access-Abfrage** ersetzen:

`MainWindow.xaml.cpp` (Anpassung für MS Access)

```
#include "pch.h"
#include "MainWindow.xaml.h"
#if __has_include("MainWindow.g.cpp")
#include "MainWindow.g.cpp"
#endif

// ODBC-Header für Datenbankzugriff einbinden
#include <windows.h>
#include <sql.h>
#include <sqlext.h>
#include <string>

using namespace winrt;
using namespace Microsoft::UI::Xaml;

namespace winrt::MeinProjekt::implementation
{
    MainWindow::MainWindow()
    {
        m_kundenListe =
winrt::single_threaded_observable_vector<MeinProjekt::Kunde>();
        InitializeComponent();
    }


winrt::Windows::Foundation::Collections::IObservableVector<MeinProjekt::Kun
de> MainWindow::KundenListe()
    {
        return m_kundenListe;
    }
```

```
    void MainWindow::DatenbankLaden_Click(IInspectable const&,
RoutedEventArgs const&)
    {
        m_kundenListe.Clear();

        // 1. ODBC Handles deklarieren
        SQLHENV hEnv = SQL_NULL_HENV;
        SQLHDBC hDbc = SQL_NULL_HDBC;
        SQLHSTMT hStmt = SQL_NULL_HSTMT;

        // 2. Verbindung zur Access-Datei (.accdb) aufbauen
        // Pfad zur Datenbankdatei anpassen!
        std::wstring connStr = L"Driver={Microsoft Access Driver (*.mdb,
*.accdb)};DBQ=C:\\Daten\\MeineDatenbank.accdb;";

        if (SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hEnv) ==
SQL_SUCCESS) {
            SQLSetEnvAttr(hEnv, SQL_ATTR_ODBC_VERSION,
(SQLPOINTER)SQL_OV_ODBC3, 0);

            if (SQLAllocHandle(SQL_HANDLE_DBC, hEnv, &hDbc) == SQL_SUCCESS)
{
                SQLWCHAR outConnStr[1024];
                SQLSMALLINT outConnStrLen;

                // Verbindung herstellen
                SQLRETURN ret = SQLDriverConnect(hDbc, NULL,
(SQLWCHAR*)connStr.c_str(), SQL_NTS,
                                                 outConnStr, 1024,
&outConnStrLen, SQL_DRIVER_NOPROMPT);

                if (ret == SQL_SUCCESS || ret == SQL_SUCCESS_WITH_INFO) {
                    // 3. SQL-Statement vorbereiten und ausführen
                    if (SQLAllocHandle(SQL_HANDLE_STMT, hDbc, &hStmt) ==
SQL_SUCCESS) {
                        std::wstring sql = L"SELECT Id, Name, Email FROM
Kunden";

                        if (SQLExecDirect(hStmt, (SQLWCHAR*)sql.c_str(),
SQL_NTS) == SQL_SUCCESS) {

                            // Puffer für die Spaltendaten definieren
                            int32_t id = 0;
                            wchar_t name[255] = { 0 };
                            wchar_t email[255] = { 0 };
                            SQLLEN lenId = 0, lenName = 0, lenEmail = 0;

                            // 4. Datenzeilen aus Access auslesen (Fetch-
Schleife)
                            while (SQLFetch(hStmt) == SQL_SUCCESS) {
                                SQLGetData(hStmt, 1, SQL_C_LONG, &id, 0,
&lenId);
                                SQLGetData(hStmt, 2, SQL_C_WCHAR, name,
sizeof(name), &lenName);
                                SQLGetData(hStmt, 3, SQL_C_WCHAR, email,
sizeof(email), &lenEmail);

                                // Das WinRT/COM-Objekt für die WinUI-
Oberfläche erstellen
                                auto neuerKunde =
winrt::make<winrt::MeinProjekt::implementation::Kunde>(
                                    id,
                                    winrt::hstring(name),
```

```
                                    winrt::hstring(email)
                                );

                                // Zeile der UI-Liste hinzufügen
(Aktualisiert das XAML sofort automatisch)
                                m_kundenListe.Append(neuerKunde);
                            }
                        }
                        SQLFreeHandle(SQL_HANDLE_STMT, hStmt);
                    }
                }
                SQLDisconnect(hDbc);
                SQLFreeHandle(SQL_HANDLE_DBC, hDbc);
            }
            SQLFreeHandle(SQL_HANDLE_ENV, hEnv);
        }
    }
}
```

Was passiert hier genau?

1. **`Driver={Microsoft Access Driver...}`**: Teilt Windows mit, dass die offizielle Access-Engine geladen werden soll.
2. **`DBQ=C:\...`**: Der absolute Pfad zu Ihrer `.accdb`-Datei. (Hinweis: Da WinUI 3 Apps verpackt sein können, achten Sie darauf, dass die App Zugriff auf diesen Ordnerpfad hat, oder nutzen Sie den Dokumenten-Ordner des Nutzers).
3. **`SQLFetch`**: Läuft so lange durch die Access-Tabelle, bis alle Zeilen gelesen sind.
4. **`winrt::make`**: Wandelt die rohen C++ Daten (`wchar_t`) in das WinUI-kompatible COM-Objekt um, das wir im vorherigen Schritt in der **IDL-Datei** definiert haben. [2]

Möchten Sie als Nächstes wissen, wie Sie Daten zurück in die Access-Datenbank schreiben (`INSERT` / `UPDATE` per ODBC), oder gibt es Fragen zur Konfiguration der **64-Bit/32-Bit-Treiber**?

Wie befürchtet, statt ADO, ODBC, statt MFC, der alte Quellcode

Ihre Enttäuschung ist absolut verständlich und trifft den Nagel auf den Kopf. Es fühlt sich paradox an: Man möchte eine hochmoderne Benutzeroberfläche mit WinUI 3 bauen, wird aber beim Datenbankzugriff in C++ scheinbar um Jahrzehnte zurückgeworfen und muss sich mit rohem ODBC-C-Code herumschlagen, der an alte MFC- oder gar Vor-MFC-Zeiten erinnert.

Der Grund für dieses Dilemma ist historisch: Microsoft hat C++ im Bereich der klassischen Desktop-Datenbanken (wie Access) seit dem Ende von MFC und dem klassischen ADO (ActiveX Data Objects) stark vernachlässigt. Das moderne ADO.NET und das Entity Framework wurden exklusiv für C# entwickelt.

Sie müssen diesen "alten Quellcode" jedoch **nicht** so hinnehmen. Es gibt zwei Wege, wie Sie den C++ Code modernisieren und den ODBC-Ballast komplett verstecken können.

Option 1: Die "Modern C++" Lösung (Kapselung über Wrapper)

Man schreibt rohes ODBC heute nicht mehr direkt in die UI-Logik. Stattdessen nutzt man schlanke C++17/C++20 Wrapper, die Ihnen die Speicherverwaltung (RAII) und die Pointer-Arbeit abnehmen. Eine der bekanntesten und einfachsten Open-Source-Bibliotheken dafür ist **NANODBC**.

Mit einem solchen modernen Wrapper schrumpft der gesamte Access-Datenbankzugriff in Ihrem WinUI 3-Projekt auf wenige, saubere Zeilen zusammen:

```
#include <nanodbc/nanodbc.h>

void MainWindow::DatenbankLaden_Click(IInspectable const&, RoutedEventArgs
const&)
{
    m_kundenListe.Clear();

    // Moderne C++ Verbindung (RAII verwaltet die Handles automatisch)
    std::wstring connStr = L"Driver={Microsoft Access Driver (*.mdb,
*.accdb)};DBQ=C:\\Daten\\MeineDatenbank.accdb;";
    nanodbc::connection connection(connStr);

    // Einfache Ausführung und Iteration wie in modernen Sprachen
    auto results = nanodbc::execute(connection, L"SELECT Id, Name, Email
FROM Kunden");

    while (results.next()) {
        // Sicheres Auslesen ohne manuelle Puffergrößen
        int32_t id = results.get<int>(0);
        std::wstring name = results.get<std::wstring>(1);
        std::wstring email = results.get<std::wstring>(2);

        // Erstellen des WinUI-Objekts (IDL)
        auto neuerKunde =
winrt::make<winrt::MeinProjekt::implementation::Kunde>(
            id, winrt::hstring(name), winrt::hstring(email)
        );
        m_kundenListe.Append(neuerKunde);
    }
}
```

Vorteil: Kein einziges `SQLAllocHandle`, kein `SQLFetch` und keine kryptischen Fehlercodes mehr. Wenn etwas schiefgeht, wirft die Bibliothek eine moderne C++-Exception.

Option 2: Die radikale, aber oft beste Alternative (C++/CLI oder C#-Hybrid)

Wenn Sie den Komfort von ADO.NET (wie Sie es vielleicht von früher kennen) oder modernen OR-Mappern vermissen, gibt es in der WinUI 3-Welt einen legitimen Architektur-Kniff: **Die App-Logik in C# schreiben, die Performance-Kerne in C++.**

Da WinUI 3, das Windows App SDK und die IDL-Schnittstelle genau dafür gebaut wurden (Sprachunabhängigkeit dank COM), können Sie:

1. Das Hauptprojekt in **C#** anlegen. Dort nutzen Sie den extrem komfortablen C#-Datenbankzugriff für Access (z.B. über `System.Data.OleDb` oder ADO.NET).
2. Eine **C++ Windows-Runtime-Komponente** (DLL) zum Projekt hinzufügen.

3. Über die IDL-Schnittstelle die rechenintensiven Algorithmen oder Altdaten-Logiken in performantem C++ ausführen lassen. [1, 2, 3]

Das XAML-System merkt keinen Unterschied, ob die Daten aus C# oder C++ kommen, da am Ende ohnehin alles über dieselbe WinRT-Infrastruktur läuft.

Welcher dieser beiden Ansätze würde Ihnen das Leben für Ihr Projekt leichter machen? Soll ich Ihnen zeigen, wie Sie eine moderne Bibliothek wie **nanodbc** unkompliziert über den Visual Studio Paketmanager (vcpkg/NuGet) einbinden, oder klingt die **Hybrid-Architektur (C# für Daten, C++ für Logik)** für Sie interessanter?

Kein C# erstmal, Tipp für IDL, welches Werkzeug funktioniert in Visual Studio 2022, 2026. Ich habe seit Jahren Fehler in dem IDL Werkzeug in Visual Studio gemeldet. COM Entwicklung wurde ja massiv eingeschränkt

Dass Sie frustriert über die IDL-Werkzeuge in Visual Studio sind, ist mehr als berechtigt. Der klassische MIDL-Compiler (`midl.exe`) und das dazugehörige Syntax-Highlighting in Visual Studio fühlen sich seit Jahren vernachlässigt an. Die Autovervollständigung (IntelliSense) streikt oft, Fehlermeldungen sind kryptisch und gemeldete Bugs in der IDE-Integration bleiben leider häufig ungelöst, da Microsoft den Fokus stark auf C# verschoben hat.

Für die moderne WinUI 3-Entwicklung mit C++/WinRT gibt es jedoch einen entscheidenden Unterschied zu früher: Es wird der neuere **MIDL 3.0 Compiler** (`midl.exe` mit dem Schalter `/winrt`) genutzt. Dieser unterscheidet sich grundlegend vom alten, klassischen COM-IDL.

Da Visual Studio von Haus aus hier immer noch Schwächen zeigt, helfen Ihnen die folgenden Werkzeuge, Erweiterungen und Workarounds, um in Visual Studio 2022 und 2026 stabil und fehlerfrei mit IDL zu arbeiten:

1. Die wichtigste Erweiterung: "C++/WinRT Visual Studio Extension"

Falls noch nicht geschehen, installieren Sie über den Visual Studio Marketplace (Menü: *Erweiterungen -> Erweiterungen verwalten*) die offizielle **C++/WinRT Extension**.

- **Was sie tut:** Sie liefert die Projektschnittstellen und sorgt dafür, dass der Build-Prozess die IDL-Dateien korrekt verarbeitet und die benötigten `.g.h`- und `.g.cpp`-Dateien im Hintergrund fehlerfrei generiert.
- **Tipp für die Praxis:** Wenn IntelliSense in der `.idl`-Datei "rote Wellenlinien" (Fehler) anzeigt, obwohl das Projekt fehlerfrei baut, ignorieren Sie diese. Das ist ein bekannter, chronischer Bug der IDE. Maßgeblich ist *nur* das Ergebnis des Compilers beim Build (`Strg + Umschalt + B`).

2. Bessere Syntax-Hervorhebung via VS Code (Der pragmatische Entwickler-Trick)

Da die IDL-Textbearbeitung in Visual Studio 2022/2026 oft keine korrekte Code-Einfärbung oder Klammer-Prüfung bietet, lagern viele Entwickler das Schreiben der IDL-Dateien temporär aus:

1. Öffnen Sie die `.idl`-Datei parallel in **Visual Studio Code**.
2. Installieren Sie dort die Erweiterung **"MIDL"** (von *maddox* oder ähnlichen Anbietern).
3. VS Code bietet eine hervorragende, fehlerfreie Syntax-Hervorhebung für MIDL 3.0. Sie schreiben den Code dort, speichern, und wechseln zurück zu Visual Studio, um das Projekt zu bauen.

3. IDL-Fehler umschiffen: Der "No-IDL"-Workaround (XAML-freie Komponenten)

Da Sie das IDL-Werkzeug verständlicherweise meiden möchten, gibt es einen architektonischen Trick für Ihre Access-Datenbankanwendung: **Schreiben Sie so wenig IDL wie möglich.**

Sie müssen *nicht* Ihre gesamte Datenbank-Logik in Klassen packen, die in der IDL deklariert sind. Nutzen Sie IDL **ausschließlich für die reinen UI-Datenklassen** (wie das `Kunde`-Objekt aus dem vorherigen Beispiel).

- Ihre eigentliche Datenbank-Verbindung, die ODBC/Nanodbc-Aufrufe, die SQL-Queries und die Business-Logik schreiben Sie in **ganz normalen, standardmäßigen C++-Klassen** (`.h` und `.cpp`), die überhaupt nicht in der IDL auftauchen.
- Erst wenn die Daten fertig geladen sind, mappen Sie die nativen C++-Strukturen in die wenigen, einfachen IDL-Datenklassen um, damit das XAML sie anzeigen kann. So reduzieren Sie den IDL-Aufwand auf ein absolutes Minimum (oft nur eine einzige Datei für Datenmodelle).

4. Das "Clean & Rebuild" Geheimnis bei Generierungs-Fehlern

Wenn der C++/WinRT-Compiler nach Änderungen an der IDL-Datei meldet, dass er die Implementierungsdateien nicht finden kann oder unauflösbare Linker-Fehler wirft, liegt das meist an einem hängengebliebenen Cache im `Generated Files`-Ordner.

- **Lösung:** Klicken Sie mit der rechten Maustaste auf die `.idl`-Datei, wählen Sie **Kompilieren** (`Strg + F7`). Wenn das fehlschlägt, führen Sie ein **Projekt bereinigen** durch und löschen Sie manuell die Ordner `x64` (oder `x86`) sowie `Generated Files` im Projektverzeichnis, bevor Sie neu bauen. Das erzwingt eine saubere Neu-Generierung der COM-Schnittstellen.

Soll ich Ihnen zeigen, wie Sie eine **normale C++-Klasse (völlig ohne IDL)** aufsetzen, die die Access-Daten abruft und diese dann gesammelt an die UI übergibt, um das IDL-Werkzeug so selten wie möglich nutzen zu müssen?