Slide 1 – Chiamate native con P/Invoke
La piattaforma .NET consente di chiamare codice nativo (non gestito) tramite **Platform Invocation (P/Invoke)**.
Questo meccanismo permette a C# di invocare funzioni C/C++ esportate in librerie DLL esterne.
È utile per utilizzare API di sistema (es. Win32) o librerie native esistenti senza doverle riscrivere in codice gestito.

Slide 2 – DllImport: dichiarare funzioni esterne
Per chiamare una funzione nativa da C#, si dichiara un metodo *extern* con l'attributo `[DllImport]`.
Bisogna specificare il nome della DLL contenente la funzione e definire in C# una firma compatibile con quella nativa.
La funzione si dichiara `static extern` (tipicamente in una classe static). Esempio di importazione di `MessageBox` da Windows:

```
[DllImport("user32.dll")]
public static extern int MessageBox(IntPtr hWnd, string text, string
caption, uint type);
// Importa la funzione Win32 MessageBox dalla DLL user32.dll
// La firma C# deve corrispondere ai parametri originali (HWND, LPCTSTR,
LPCTSTR, UINT)
```

Slide 3 – Parametri di DllImport e convenzioni
L'attributo `[DllImport]` accetta parametri opzionali importanti per adattare la chiamata:
- **CharSet** – Definisce come marshalerà le stringhe: ad esempio `CharSet.Unicode` o `CharSet.Ansi`. Su Windows moderno, usare Unicode garantisce supporto ai caratteri Unicode (es. accenti).
- **CallingConvention** – Specifica la convenzione di chiamata usata dalla funzione nativa. Per le API Win32 standard è `CallingConvention.StdCall` (default); per librerie C custom spesso `CallingConvention.Cdecl`. Deve combaciare con la convenzione usata nella DLL.
- **SetLastError** – Se impostato a true, indica al runtime di salvare il risultato di `GetLastError()` dopo la chiamata. Può essere recuperato in C# con `Marshal.GetLastWin32Error()`. Utile per funzioni Win32 che segnalano errori tramite `GetLastError`.

Slide 4 – Marshaling dei tipi base
Il *marshaling* è il processo di conversione dei tipi tra il mondo managed (.NET) e unmanaged (C/C++).
Alcuni tipi primitivi sono *blittable*: hanno la stessa rappresentazione in memoria in C# e C++ e vengono passati così come sono (es. `int`, `double`, `byte`, `IntPtr`).
Altri tipi richiedono conversione: ad esempio, il `bool` di .NET (1 byte) viene convertito nel `BOOL` Win32 (4 byte) automaticamente; il `char` .NET (Unicode 16-bit) può diventare `char` ANSI 8-bit se specificato.
Il marshaller .NET gestisce automaticamente le conversioni standard, ma è importante dichiarare correttamente i tipi (es.: usare `IntPtr` o `UInt32` per puntatori/handle nativi, a seconda dei casi).

Slide 5 – Marshaling di stringhe
Le stringhe in C# vengono convertite in puntatori a caratteri C durante la chiamata P/Invoke.
Di default, se `CharSet=Unicode` (o Auto su Windows), una `string` .NET viene passata come

puntatore a stringa Unicode (LPWSTR) terminata da null. Con `CharSet.Ansi`, viene convertita in ANSI (LPSTR) per funzioni che lo richiedono.
Per ricevere output in una stringa, spesso si usa un buffer mutabile: ad esempio passare un oggetto `StringBuilder` inizializzato a una certa lunghezza, che il codice nativo riempirà con testo.
In alternativa, si può ottenere un puntatore non gestito (IntPtr) di output e convertirlo con `Marshal.PtrToStringAnsi/Uni`. In tal caso, bisogna anche occuparsi di liberare la memoria se il puntatore è allocato dal lato nativo (es. con `Marshal.FreeHGlobal` o funzioni di deallocazione specifiche).

Slide 6 – Marshaling di strutture
Le strutture C/C++ possono essere passate a P/Invoke definendo una struct equivalente in C#.
Bisogna dichiararla con `[StructLayout(LayoutKind.Sequential)]` per mantenere l'ordine e l'allineamento dei campi come nel codice nativo. I tipi dei campi devono corrispondere (es.: `int` per `LONG` in Win32, `short` per `SHORT`, etc.).
Si può passare la struct come parametro `ref` o `out` se la funzione nativa richiede un puntatore alla struttura. In caso di struct restituita, si può usare il tipo come return (il marshaller copierà i valori).
Esempio: definizione di una struct `POINT` per usare la funzione Win32 `GetCursorPos` che riempie una POINT:

```
[StructLayout(LayoutKind.Sequential)]
public struct POINT { public int X; public int Y; }  // struttura C#
compatibile con POINT nativa (due LONG)
[DllImport("user32.dll")]
static extern bool GetCursorPos(out POINT lpPoint); // dichiara GetCursorPos
che scrive le coordinate in POINT
// ...
POINT pos;
GetCursorPos(out pos);  // chiama la funzione, riempiendo la struct pos
Console.WriteLine($"{pos.X}, {pos.Y}");  // usa i dati ottenuti
```

Slide 7 – Esempio: chiamata a funzione Win32
Una volta importata la funzione nativa, la si usa come un normale metodo statico. Ad esempio, utilizziamo la `MessageBox` importata per visualizzare una finestra di messaggio:

```
int result = MessageBox(IntPtr.Zero, "Ciao dal P/Invoke!", "Saluto",
1); // 1 = MB_OKCANCEL, mostra OK/Cancel
if(result == 1) {
    Console.WriteLine("Premuto OK");     // ID 1 corrisponde al tasto "OK"
} else {
    Console.WriteLine("Premuto Annulla"); // ID 2 tipicamente per "Cancel"
}
```

In questo esempio, passiamo `IntPtr.Zero` perché la MessageBox non ha una finestra proprietaria (hWnd = 0). Il marshaller converte le stringhe `.NET` in stringhe native secondo il CharSet impostato (qui default Ansi se non specificato). Il risultato `int` indica quale pulsante è stato premuto.

Slide 8 – Errori comuni con P/Invoke
- **Firma non corrispondente:** se i tipi o la convenzione di chiamata dichiarati in C# non corrispondono

esattamente a quelli della funzione nativa, si possono avere crash (es. `AccessViolationException`) o risultati errati. Esempio tipico: usare `int` al posto di `IntPtr` per un handle o puntatore su sistemi 64-bit causa una dimensione errata.
- **Marshalling errato di stringhe:** dimenticare di specificare il CharSet può portare a testo illeggibile o caratteri incoerenti (Unicode vs ASCII).
- **Memory leak:** se la funzione nativa alloca memoria (ad es. restituisce un puntatore a buffer) e il codice C# non la libera, si ha perdita di memoria. Il marshaller non sa liberare automaticamente memoria allocata dal codice nativo (a meno che sia un BSTR o memoria COM). Bisogna chiamare esplicitamente la funzione di deallocazione appropriata o usare `Marshal.FreeHGlobal/CoTaskMem` se applicabile.
- **Accesso a memoria non valida:** passare riferimenti errati (es. puntatore null non previsto, buffer troppo piccolo) o definire struct sbagliate può portare a leggere/scrivere memoria non valida e causare crash dell'applicazione.

Slide 9 – Best practice P/Invoke
- Definire con precisione la firma: controllare documentazione della funzione nativa e usare tipi C# adeguati (es.: `UInt32` per `DWORD`, `IntPtr` per puntatori o handle, `Int16`/`Int64` per tipi di larghezza specifica). Impostare sempre la `CallingConvention` corretta se diversa dallo stdcall predefinito.
- Specificare il `CharSet` nell'attributo `[DllImport]` per chiarire la conversione delle stringhe (evita ambiguità e avvisi, es. `CharSet.Unicode` per funzioni Unicode). Utilizzare `[MarshalAs]` su parametri se necessario per forzare un certo tipo di marshaling (es. `UnmanagedType.LPStr` o `LPWStr`).
- Gestire la memoria cross-boundary: se una funzione nativa richiede che il chiamante liberi la memoria (o viceversa), assicurarsi di farlo. Ad esempio, se la DLL fornisce una funzione `FreeXyz()` per liberare una struttura allocata da `GetXyz()`, chiamarla sempre. In mancanza, utilizzare le funzioni di marshalling (`Marshal.FreeHGlobal`, etc.) appropriate.
- Usare **SafeHandle** per risorse native quando possibile: creando una classe derivata da `SafeHandle` si delega al framework la liberazione dell'handle (nel metodo `ReleaseHandle`). Questo evita errori di mancato rilascio di risorse e integra Dispose/using pattern.
- Minimizzare le chiamate ripetitive: l'invocazione P/Invoke ha un costo fisso; se bisogna chiamare in loop migliaia di volte una funzione nativa, può essere inefficiente. Meglio valutare di raggruppare i dati e fare un'unica chiamata che esegua più lavoro a livello nativo, oppure spostare il loop nel codice non gestito se possibile.

Slide 10 – Conclusioni P/Invoke
P/Invoke è uno strumento potente per estendere applicazioni .NET con funzionalità native.
Permette di riutilizzare librerie C/C++ esistenti e accedere a chiamate di sistema non presenti nelle librerie .NET.
Va utilizzato con attenzione: una configurazione errata può causare errori difficili da diagnosticare. Con una corretta definizione delle funzioni e una gestione appropriata di stringhe, strutture e memoria, P/Invoke offre un'integrazione fluida tra codice managed C# e funzioni native.
In caso di necessità più complesse (molte funzioni, oggetti, callback), si possono considerare anche altre forme di interop come COM o C++/CLI, di cui parleremo nei prossimi blocchi.

Slide 11 – Introduzione a COM Interop
COM (Component Object Model) è un modello di componenti software introdotto da Microsoft prima di .NET, basato su oggetti e interfacce binarie.
Molte funzionalità di Windows e applicazioni (es. Microsoft Office) espongono **oggetti COM** per automatizzare operazioni.
COM Interop in .NET permette a C# di creare e usare oggetti COM come fossero oggetti .NET, grazie a

un wrapper che si occupa di tradurre chiamate e tipi tra il runtime .NET e COM. Questo consente di integrare componenti COM esistenti (ad esempio, usare Excel o Word via automazione) nelle applicazioni C#.

Slide 12 – Registrazione dei componenti COM
Per essere utilizzato, un componente COM deve essere registrato nel sistema. La registrazione (tipicamente tramite `regsvr32` per componenti nativi) crea voci nel Registro di Windows che associano un CLSID (GUID) e un ProgID leggibile a classe e interfacce esposte.
Una volta registrato, un oggetto COM può essere istanziato dalle applicazioni tramite il suo ProgID o CLSID.
Anche librerie .NET possono essere esposte come COM (con attributi `[ComVisible(true)]` e `[Guid]` sui tipi, e usando `regasm` per registrarle): questo crea un *COM Callable Wrapper* permettendo a codice C++ (unmanaged) di richiamare classi .NET come COM. (Viceversa del Runtime Callable Wrapper usato da C# per chiamare COM).

Slide 13 – Utilizzare un oggetto COM in C#
In C#, per usare un oggetto COM bisogna ottenere un wrapper .NET dell'oggetto COM, chiamato **RCW (Runtime Callable Wrapper)**.
Ci sono due modi principali:
- **Importazione anticipata (early binding):** aggiungere un riferimento COM al progetto (Visual Studio genererà un assembly di Interop contenente definizioni di interfacce e classi COM). Dopodiché si può usare l'oggetto con le sue interfacce tipizzate come fossero classi .NET.
- **Creazione dinamica (late binding):** usare `Type.GetTypeFromProgID` o `Type.GetTypeFromCLSID` per ottenere il `System.Type` associato a un ProgID/CLSID registrato, quindi creare un'istanza con `Activator.CreateInstance`. Questo restituisce un `object` (o dynamic) che si interfaccia con l'oggetto COM. Si potranno invocare i membri tramite riflessione o usando `dynamic` (che usa IDispatch).

In entrambi i casi, .NET dietro le quinte gestisce le chiamate COM: il RCW converte tipi (stringhe, numeri, oggetti) e richiama i metodi COM tramite le vTable delle interfacce. Il codice C# può quindi interagire con COM quasi trasparentemente.

Slide 14 – Importare librerie COM con Visual Studio
Visual Studio semplifica l'integrazione COM tramite l'**aggiunta di riferimenti** COM:
- Nel progetto C#, si sceglie "Add Reference..." e nella scheda COM si seleziona la libreria o componente desiderato (deve essere registrato nel sistema).
- VS utilizza lo strumento tlbimp (Type Library Importer) per generare un assembly di interop .NET contenente definizioni di classi e interfacce COM in termini di tipi .NET (ad esempio verrà creato un file Interop.NomeLibreria.dll).
- Una volta aggiunto il riferimento, nel codice C# si possono istanziare oggetti COM come se fossero classi C#. In realtà vengono creati via COM, ma grazie all'RCW possiamo accedere a proprietà e metodi definiti dall'interfaccia COM.

Esempio: aggiungendo il riferimento a **Microsoft Excel Object Library**, possiamo fare `var excel = new Microsoft.Office.Interop.Excel.Application();` in C# per ottenere l'istanza COM di Excel.

Slide 15 – Importare librerie COM con tlbimp
In alternativa a Visual Studio, si può usare direttamente lo strumento a riga di comando **tlbimp.exe** per generare l'assembly di interop da una libreria di tipi COM:

```
tlbimp.exe MiaLibreria.tlb /out=MiaLibreria.Interop.dll
```

Questo comando legge la type library `MiaLibreria.tlb` (che descrive i coclass e le interfacce COM esposte) e crea un assembly .NET chiamato *MiaLibreria.Interop.dll*.
Tale assembly conterrà le classi e interfacce COM trasformate in equivalenti .NET (interfacce diventano interfacce .NET, coclass diventano classi).
Dopodiché, nel progetto C# basterà referenziare **MiaLibreria.Interop.dll** e si potranno usare i tipi come definiti. Il runtime si occuperà di creare gli oggetti COM reali quando si istanzia una di queste classi (chiamando CoCreateInstance internamente).

Slide 16 – Esempio: Automazione di Excel con COM
Mostriamo un esempio pratico di utilizzo di un oggetto COM esistente: Excel. Tramite COM Interop possiamo lanciare Excel e manipolarlo. *(Si assume di aver aggiunto il riferimento a Microsoft Excel Interop nel progetto).*

```
using Excel = Microsoft.Office.Interop.Excel;            // Alias per lo
spazio dei nomi di Excel Interop
Excel.Application xlApp = new Excel.Application();       // Crea una nuova
istanza dell'applicazione Excel
xlApp.Visible = true;                                    // Rende Excel
visibile all'utente
Excel.Workbook wb = xlApp.Workbooks.Add();               // Aggiunge una
nuova cartella di lavoro (Workbook)
Excel.Worksheet foglio = (Excel.Worksheet)wb.Sheets[1]; // Ottiene il primo
foglio (Worksheet) del workbook
foglio.Cells[1, 1].Value = "Ciao";                       // Scrive il valore
"Ciao" nella cella A1
```

In questo codice, ogni oggetto (`Application`, `Workbook`, `Worksheet`) è un wrapper .NET attorno all'oggetto COM reale. Possiamo accedere a proprietà/metodi (come `Workbooks.Add` o `Cells.Value`) come se fossero oggetti C#, ma in realtà queste chiamate vengono inoltrate via COM ad Excel.

Slide 17 – Esempio: utilizzo di un oggetto COM via ProgID
Senza un assembly di interop predefinito, possiamo usare la creazione dinamica. Ad esempio, utilizziamo il **FileSystemObject** (COM per gestire il filesystem, fornito da Windows Scripting):

```
Type fsoType = Type.GetTypeFromProgID("Scripting.FileSystemObject");  //
Ottiene il Type COM dal ProgID
dynamic fso =
Activator.CreateInstance(fsoType);                       // Crea un'istanza
dell'oggetto COM (late binding)
var driveC = fso.GetDrive("C:");                                       //
Chiama un metodo COM (ritorna oggetto Drive)
Console.WriteLine($"Disco C: {driveC.FreeSpace} bytes liberi");       //
Legge una proprietà COM dall'oggetto ritornato
```

Qui `Type.GetTypeFromProgID` cerca nel registro il ProgID `"Scripting.FileSystemObject"` e trova il CLSID corrispondente. `Activator.CreateInstance` effettua internamente CoCreateInstance sul CLSID e ci restituisce un RCW. Usando `dynamic`, chiamiamo metodi e proprietà senza dover avere definizioni forti a compile-time: il runtime usa IDispatch per invocare `GetDrive` e ottenere la proprietà `FreeSpace`. Questo approccio è utile per script o casi dove non si vuole/prevede un interop assembly.

Slide 18 – Differenze tra P/Invoke e COM Interop
- **Tipo di interazione:** P/Invoke chiama funzioni in stile C esposte da DLL (interfaccia procedurale), mentre COM Interop lavora con oggetti e interfacce (approccio orientato agli oggetti).
- **Preparazione:** P/Invoke richiede di conoscere e definire in C# la firma esatta di ogni funzione chiamata (dettagli di parametri, tipi, convenzioni). Con COM, è sufficiente avere la libreria registrata o una libreria dei tipi: le definizioni vengono importate automaticamente (riducendo errori di firma).
- **Marshalling e gestione errori:** Con P/Invoke il marshalling è determinato dai tipi primitivi e attributi (lo sviluppatore deve occuparsi di casi particolari). Il codice nativo comunica errori via codici di ritorno o `GetLastError`. In COM Interop, il marshaller .NET converte automaticamente tipi COM (BSTR per stringhe, VARIANT per oggetti) in tipi .NET. Inoltre, errori COM (HRESULT) vengono automaticamente trasformati in eccezioni .NET (`COMException` o specifiche se registrate).
- **Gestione della memoria e delle risorse:** P/Invoke non fornisce meccanismi integrati di gestione risorse – il codice .NET deve assicurarsi di rilasciare risorse native. In COM, ogni oggetto ha reference counting: il RCW incrementa il conteggio COM quando si crea l'oggetto e lo decrementa quando viene rilasciato/garbage collectato. Si può forzare il rilascio immediato chiamando `Marshal.ReleaseComObject`. Inoltre, P/Invoke può chiamare qualsiasi libreria nativa (non richiede registrazione), mentre COM richiede componenti registrati globalmente (o registration-free manifest).

Slide 19 – Best practice con Interop COM
- **Rilascio degli oggetti:** Non aspettare il garbage collector per liberare oggetti COM pesanti (es. oggetti Excel/Word). Usa `Marshal.ReleaseComObject(obj)` quando hai finito con un oggetto, specialmente in automazione Office per evitare che Excel rimanga aperto. In loop, rilascia gli oggetti temporanei (es. Range in Excel) per non accumulare reference count.
- **Threading COM:** Assicurati che il thread che utilizza l'oggetto COM sia nel contesto giusto. Molti componenti COM richiedono **STA (Single Threaded Apartment)**. In un'applicazione .NET Windows Forms/WPF il thread principale è STA per default, ma in una console app potrebbe essere MTA: in tal caso, usa `[STAThread]` sull'entry point o crea thread STA tramite `Thread.SetApartmentState` prima di usare COM come Excel (che richiede STA).
- **Gestione eccezioni:** Le chiamate COM possono lanciare eccezioni (`COMException`). Usa try/catch attorno a operazioni COM per gestire errori (ad esempio fallimenti di instanziazione, metodi non supportati, errori specifici dell'app COM). Puoi controllare `ex.ErrorCode` (HRESULT) per dettagli sull'errore.
- **Prestazioni e design:** Limitare chiamate COM in loop critici (simile a P/Invoke overhead). Se devi invocare molte volte un metodo COM, valuta di ridurre i passaggi (ad es. invece di leggere una cella Excel alla volta in un loop .NET, usa operazioni in blocco come assegnare un array a `Range.Value`). Inoltre, preferisci usare interfacce duali o dispinterfaces dirette se possibile, per evitare overhead di IDispatch (late binding) – ovvero, utilizza i tipi forti forniti dall'assembly Interop quando li hai.

Slide 20 – Conclusioni COM Interop
L'integrazione con COM consente alle applicazioni .NET di sfruttare una vasta gamma di componenti esistenti, specialmente in ambiente Windows (Office, componenti di sistema, librerie legacy).
COM Interop offre un ponte relativamente semplice: dopo la registrazione e import, gli oggetti COM appaiono quasi come oggetti locali .NET.

Bisogna però considerare i vincoli di COM: la necessità di registrare i componenti (o usare manifest per registration-free COM), la dipendenza da Windows (COM è nativo di Windows), e la gestione del ciclo di vita (rilascio manuale se necessario per evitare memory leak o lock di applicazioni COM).
In sintesi, COM Interop è ideale quando si deve interagire con software esistente in forma di componenti, mentre per nuove librerie native spesso si preferisce P/Invoke o altre soluzioni. Nel prossimo blocco vedremo altri scenari e approcci di interoperabilità avanzata (C++/CLI, chiamate bidirezionali, ecc.).

Slide 21 – Interoperabilità avanzata: scenari comuni
In progetti reali, non è raro dover far cooperare codice managed (.NET) e unmanaged (C/C++). Alcuni scenari tipici:
- **Riutilizzo di librerie native esistenti:** ad esempio motori di calcolo scientifico, librerie grafiche o di database scritte in C/C++. Invece di riscriverle in C#, si integrano via interop (P/Invoke o COM se sono COM, o wrapper dedicati).
- **Accesso a funzionalità di sistema/os-specific:** molte funzioni del sistema operativo o API di basso livello (driver, chiamate di sistema) non hanno un wrapper .NET pronto. Si ricorre a P/Invoke per chiamare direttamente l'API (es: chiamare funzioni Win32 per manipolare registri di sistema, gestire memoria, ecc.).
- **Performance-critical code:** parti dell'applicazione con esigenze di alta performance possono beneficiare dell'implementazione in C++ nativo (più efficiente, controlla manuale della memoria). Si può usare C++ per la parte intensiva e chiamarla da C#, ottenendo il risultato più velocemente rispetto a un'implementazione gestita pura, soprattutto in calcoli numerici, elaborazione di immagini/audio/video, algoritmi crittografici, ecc.

Slide 22 – Esempio: chiamare una libreria C via P/Invoke
Supponiamo di avere una libreria nativa C/C++ (ad esempio *MathLib.dll*) che espone funzioni utili. Possiamo usare P/Invoke per chiamarle dal nostro codice C#.

```
[DllImport("MathLib.dll", CallingConvention=CallingConvention.Cdecl)]
static extern double CalcolaMedia(double a, double b);  // Import della
funzione nativa: double CalcolaMedia(double,double)

double risultato = CalcolaMedia(5.0, 7.5);             // Chiamata della
funzione nativa con due parametri double
Console.WriteLine($"Media calcolata: {risultato}");     // Usa il risultato
restituito (ad es. 6.25 per 5.0 e 7.5)
```

In questo esempio, la funzione `CalcolaMedia` viene chiamata come se fosse un metodo C# qualsiasi. Il runtime si occupa di passare i valori double alla DLL e restituire il double risultato. Notare l'uso di `CallingConvention.Cdecl` perché la libreria C++ ipoteticamente usa la convenzione cdecl (tipica delle funzioni C non standard).

Slide 23 – C++/CLI: un ponte tra C# e C++
Quando le interazioni con il codice nativo diventano numerose o complesse (ad esempio una libreria orientata agli oggetti, con molte funzioni e strutture), P/Invoke puro può diventare oneroso.
C++/CLI è una variante di C++ che consente di scrivere codice managed e unmanaged nello stesso file, fungendo da **ponte** tra i due mondi.
Con C++/CLI si può creare un *wrapper* managed attorno a codice C++ nativo: in pratica, si implementano classi .NET (ref class) i cui metodi chiamano internamente funzioni o metodi C++ nativi. Il risultato è compilato in un assembly .NET (DLL o EXE) che espone queste classi wrapper a C#.

Vantaggi: possibilità di incapsulare logiche C++ complesse, evitare di scrivere decine di dichiarazioni P/Invoke in C#, e performance migliori nelle chiamate ripetute (le chiamate dal C++/CLI al C++ nativo sono dirette, senza overhead di P/Invoke per ogni invocazione).

Slide 24 – Esempio: Wrapper C++/CLI
Di seguito un esempio semplificato di wrapper in C++/CLI. Immaginiamo una classe C++ nativa `NativeCalc` con un metodo Somma; creiamo una classe managed `CalcWrapper` che la incapsula:

```
// C++/CLI - Esempio di wrapper gestito attorno a una classe C++ nativa
class NativeCalc {                          // Classe C++ nativa esposta da una libreria
public:
    int Somma(int a, int b) { return a + b; }
};

public ref class CalcWrapper {              // Classe gestita (ref class) visibile a C#
private:
    NativeCalc* nativa;                     // Puntatore alla implementazione nativa
public:
    CalcWrapper() { nativa = new NativeCalc(); }            // Costruttore: crea l'istanza nativa
    int Somma(int a, int b) { return nativa->Somma(a, b); } // Metodo managed che richiama il metodo nativo
    ~CalcWrapper() { delete nativa; }                       // Distruttore deterministico (Dispose): libera risorse native
    !CalcWrapper() { delete nativa; }                       // Finalizer (backup nel GC) nel caso Dispose non sia chiamato
};
```

Compilando questo codice in un assembly .NET (con /clr), otteniamo una DLL utilizzabile in C#. Ad esempio, in C# potremmo fare:

```
CalcWrapper cw = gcnew CalcWrapper();           // Creazione dell'istanza gestita (gcnew è usato in C++/CLI; in C# basta 'new')
int risultato = cw.Somma(3, 4);                // Chiama internamente NativeCalc::Somma(3,4)
Console.WriteLine($"Risultato: {risultato}");   // Stampa 7
```

Dal punto di vista C#, `CalcWrapper` è una normale classe .NET (con metodi, costruttore, Dispose) ma internamente incapsula e gestisce l'oggetto C++ nativo. Questo approccio è molto potente per integrare librerie C++ complesse: tutto il dettaglio rimane nel C++/CLI, e l'uso in C# è semplice.

Slide 25 – Chiamare codice .NET da codice nativo
L'interoperabilità non è solo C# che chiama C++: a volte serve il contrario, ovvero far sì che un programma nativo invochi logica implementata in C#.
Una via è esporre componenti .NET come COM (come detto, con ComVisible e registrazione): il codice

C++ può creare istanze COM di classi .NET e usarle come se fossero oggetti COM qualsiasi. Questa strada sfrutta il **CCW (COM Callable Wrapper)** generato dal runtime .NET.
Un'altra tecnica è il **reverse P/Invoke** tramite callback: una funzione C può accettare come parametro un puntatore a funzione; dal lato C# si può definire un delegate e passarlo, e il marshaller convertirà il delegate in un function pointer compatibile. Così il codice nativo potrà chiamare "indietro" il delegate, eseguendo codice managed.

Esempio: supponiamo una libreria C++ con funzione `RegistraCallback(void (*fun)(int))` che si salva un puntatore a funzione da chiamare più tardi. Dal lato C# possiamo fare:

```
[UnmanagedFunctionPointer(CallingConvention.Cdecl)]
public delegate void Callback(int
valore);  // Delegate C# che rappresenta la firma del callback nativo

[DllImport("NativeLib.dll", CallingConvention=CallingConvention.Cdecl)]
public static extern void RegistraCallback(Callback cb);

void MioCallback(int v) { Console.WriteLine($"Callback chiamato con valore
{v}"); }

// ... registrazione del callback:
RegistraCallback(MioCallback);  // Passa il delegate; il marshaller fornisce
al nativo un puntatore a funzione
```

In questo modo, quando il codice C++ invoca la funzione puntata (es. chiama `fun(42)`), il runtime intercetta la chiamata e la inoltra al metodo C# `MioCallback(42)`. Questo meccanismo consente al codice unmanaged di eseguire routine definite nel lato managed, realizzando una comunicazione bidirezionale. (È fondamentale mantenere il delegate in vita finché può essere chiamato dal nativo, per evitare che il GC lo sposti o lo distrugga; di solito si usa un campo statico o GCHandle per mantenerlo riferenziato.)

Slide 26 – Considerazioni sulle performance
L'interazione tra codice managed e unmanaged ha un costo, dovuto al marshalling dei parametri e al cambio di contesto. In molti casi questo overhead è piccolo (microsecondi) e irrilevante per chiamate occasionali. Ma in scenari ad alte prestazioni occorre minimizzare gli overhead:
- **Ridurre il numero di chiamate interop:** Meglio chiamare una funzione nativa che elabora 1000 elementi, piuttosto che chiamarla 1000 volte per un elemento ciascuna. Batch di operazioni riducono significativamente l'impatto del passaggio tra mondi.
- **Preferire tipi semplici o già ottimali per il marshalling:** ad esempio, passare array di struct blittable è più efficiente che passare array di oggetti complessi. Si può usare `[MarshalAs(UnmanagedType.LPArray)]` per passare un array in un'unica chiamata.
- **Profilare e spostare logica se necessario:** se un loop in C# invoca tantissime volte funzioni native e diventa bottleneck, può convenire implementare l'intero loop in C++ (spostando più logica possibile nel lato nativo) e richiamarlo meno volte. Oppure, valutare l'uso di C++/CLI per chiamate frequenti, dato che una volta dentro il codice C++/CLI, si può iterare sull'oggetto nativo senza passare per il marshaller ripetutamente.

In sintesi, le chiamate interop vanno bene per interazioni di medio/basso volume; per operazioni ad altissima frequenza è meglio ripensare la suddivisione del lavoro tra managed e unmanaged.

Slide 27 – Gestione della memoria e risorse in interop
Quando si combinano codice gestito e non, è fondamentale chiarire la responsabilità della gestione delle risorse:
- **Allocazione e deallocazione memoria:** Se il codice nativo alloca memoria (ad es. restituisce un buffer), definire chi la libera. Idealmente la libreria fornisce una funzione di "Free" da chiamare via P/Invoke. Se usa memoria di sistema (CoTaskMemAlloc), si può liberare con `Marshal.FreeCoTaskMem` dal lato .NET. Viceversa, se il managed passa un buffer allocato a C++, assicurarsi che il nativo non lo conservi oltre la chiamata o documentare che deve copiarne il contenuto.
- **Pinning e GC:** Gli oggetti .NET possono essere spostati in memoria dal Garbage Collector. Se passiamo un riferimento (es. puntatore a un array) a codice nativo che lo userà in modo asincrono, bisogna "pinnare" l'oggetto (usare `GCHandle.Alloc(obj, GCHandleType.Pinned)` o parole chiave `fixed` in C#) per impedire lo spostamento finché il codice nativo ha bisogno di quell'indirizzo.
- **Risorse non gestite e SafeHandle:** Qualsiasi risorsa non gestita ottenuta tramite interop (file handle, socket nativo, device context, ecc.) dovrebbe essere incapsulata in un oggetto SafeHandle o rilasciata in un blocco finally/Dispose. Ad esempio, se chiamiamo una funzione nativa che restituisce un handle, conviene creare una classe derivata da SafeHandle per quell'handle, così che quando l'oggetto .NET viene raccolto (o Disposed) la `ReleaseHandle` invoca la CloseHandle/Free corretta. Questo riduce rischi di leak.
- **Oggetti COM:** Come detto, rilasciare esplicitamente se consumano molte risorse. Inoltre, evitare di passare oggetti COM tra thread arbitrariamente: rispettare i modelli di threading COM (se un oggetto COM è Apartment Threaded, accessibile solo dal thread che l'ha creato, usare Marshalling COM con `CoMarshalInterThreadInterfaceInStream` se davvero serve passarli, altrimenti preferire architetture che non rompano queste regole).

Slide 28 – Esempio reale: wrapping di una libreria nativa
Un caso concreto di interoperabilità è l'uso di librerie C++ di terze parti in applicazioni .NET. Ad esempio, **OpenCV** è una popolare libreria C/C++ per la visione artificiale. Per usarla in C#, esistono progetti wrapper come **Emgu CV** e **OpenCvSharp**: questi forniscono classi e metodi .NET che sotto il cofano chiamano le corrispondenti funzioni native di OpenCV.
Il pattern tipico consiste nel creare funzioni C-style esportate (o usare quelle esistenti di OpenCV) e poi utilizzare P/Invoke per ognuna, oppure scrivere un layer C++/CLI che traduce le strutture di dati (es. matrice di immagini) in tipi .NET (es. `Bitmap` o array) e chiama i metodi nativi.
Per lo sviluppatore .NET, il risultato è una libreria facile da usare (`Image img = CvInvoke.Imread("foto.jpg"); CvInvoke.Canny(img, ...);` in Emgu CV, per dire). Internamente però, ogni chiamata `CvInvoke` è un P/Invoke a `cv::imread`, `cv::Canny` etc. Questo scenario dimostra come con l'interoperabilità si possano **portare funzionalità avanzate** nel mondo .NET senza riscriverle, ma semplicemente incapsulandole.

Slide 29 – Esempio reale: codice ad alte prestazioni
Consideriamo un'applicazione che deve eseguire calcoli intensivi, ad esempio elaborare un segnale audio in tempo reale o generare grafica 3D. Pure potendo scrivere tutto in C#, potremmo incontrare limiti di performance o dover scendere a basso livello. Ecco due esempi di come si combina managed e unmanaged per performance:
- **Motori di gioco (Game Engine):** Molti game engine (Unity, Godot con Mono, ecc.) hanno il core in C/C++ per massimizzare la velocità del motore grafico e fisico. Offrono poi un layer di scripting in C# per la logica di gioco. Il collegamento avviene via interop: ad esempio, in Unity, i plugin nativi C++ vengono chiamati con P/Invoke da script C#, e il motore stesso chiama funzioni di script tramite hooking e callback interni. Così si ottiene il meglio di entrambi: core nativo ultra-performante e scripting rapido da sviluppare in C#.
- **Calcolo scientifico/algoritmi pesanti:** Librerie come Intel MKL (Math Kernel Library, in C/Fortran ottimizzato) o cuBLAS/cuDNN (GPU, C++) possono essere usate in .NET tramite interop. Un esempio è

ML.NET (machine learning in .NET) che sotto le copertine usa librerie native per operazioni matematiche pesanti via P/Invoke. Allo sviluppatore appare come chiamare funzioni C#, ma il lavoro intenso è svolto da codice nativo ottimizzato, garantendo performance elevate.

In entrambi gli esempi, l'interoperabilità è la chiave per usare componenti ad alte prestazioni scritti in linguaggi nativi all'interno di applicazioni .NET.

Slide 30 – Conclusioni sugli scenari di interoperabilità
Abbiamo esplorato diversi approcci per far comunicare C# con codice non gestito, ognuno adatto a situazioni specifiche:
- **P/Invoke:** ideale per chiamare funzioni isolate in librerie C/C++ (API di sistema o funzioni di libreria). Semplice da implementare per poche funzioni, richiede attenzione nel marshalling ma offre accesso diretto a qualunque DLL.
- **COM Interop:** utile se dobbiamo integrare componenti a oggetti già esistenti, specialmente in ambiente Windows (es. automatizzare Excel, usare controlli ActiveX, o componenti COM aziendali). La curva di integrazione è bassa se il componente è già registrato e con type library disponibile.
- **C++/CLI:** appropriato quando si dispone di una grande codebase C++ (non COM) orientata agli oggetti o con tante funzionalità: invece di scrivere centinaia di P/Invoke, si crea un layer managed in C++/CLI che espone un'API .NET pulita. Questo approccio richiede conoscenza di C++/CLI ma può risultare in un'integrazione più naturale e performante per usi estesi.

In conclusione, l'interoperabilità in .NET è un aspetto fondamentale che permette di estendere le capacità del codice gestito e riutilizzare asset esistenti. Scegliendo l'approccio giusto e seguendo le best practice (per la memoria, performance, etc.), è possibile combinare efficacemente C# e codice nativo ottenendo applicazioni potenti e flessibili.
