Walkthrough: Create and use your own Dynamic Link Library (C++)

10/06/2021 • 23 minutes to read • +4

In this article

This step-by-step walkthrough shows how to use the Visual Studio IDE to create your own dynamic link library (DLL) written in Microsoft C++ (MSVC). Then it shows how to use the DLL from another C++ app. DLLs (also known as *shared libraries* in UNIX-based operating systems) are one of the most useful kinds of Windows components. You can use them as a way to share code and resources, and to shrink the size of your apps. DLLs can even make it easier to service and extend your apps.

In this walkthrough, you'll create a DLL that implements some math functions. Then you'll create a console app that uses the functions from the DLL. You'll also get an introduction to some of the programming techniques and conventions used in Windows DLLs.

This walkthrough covers these tasks:

- Create a DLL project in Visual Studio.

- Add exported functions and variables to the DLL.
- Create a console app project in Visual Studio.
- Use the functions and variables imported from the DLL in the console app.
- Run the completed app.

Like a statically linked library, a DLL *exports* variables, functions, and resources by name. A client app *imports* the names to use those variables, functions, and resources. Unlike a statically linked library, Windows connects the imports in your app to the exports in a DLL at load time or at run time, instead of connecting them at link time. Windows requires extra information that isn't part of the standard C++ compilation model to make these connections. The MSVC compiler implements some Microsoft-specific extensions to C++ to provide this extra information. We explain these extensions as we go.

This walkthrough creates two Visual Studio solutions; one that builds the DLL, and one that builds the client app. The DLL uses the C calling convention. It can be called from apps written in other programming languages, as long as the platform, calling conventions, and linking conventions match. The client app uses *implicit linking*, where Windows links the app to the DLL at load-time. This linking lets the app call the DLL-supplied functions just like the functions in a statically linked library.

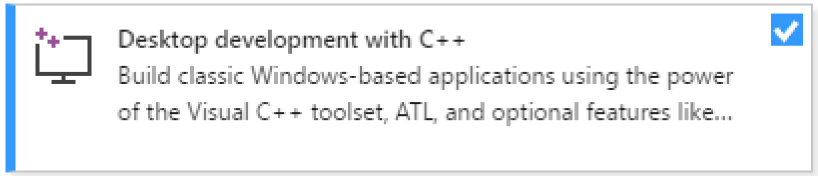
This walkthrough doesn't cover some common situations. The code doesn't show the use of C++ DLLs by other programming languages. It doesn't show how to create a resource-only DLL, or how to use explicit linking to load DLLs at run-time rather than at load-time. Rest assured, you can use MSVC and Visual Studio to do all these things.

For links to more information about DLLs, see Create C/C++ DLLs in Visual Studio. For more information about implicit linking and explicit linking, see Determine which linking method to use. For information about creating C++ DLLs for use with programming languages that use C-language linkage conventions, see Exporting C++ functions for use in C-language executables. For information about how to create DLLs for use with .NET languages, see Calling DLL Functions from Visual Basic Applications.

Prerequisites

- A computer that runs Microsoft Windows 7 or later versions. We recommend the latest version of Windows for the best development experience.

- A copy of Visual Studio. For information on how to download and install Visual Studio, see Install Visual Studio. When you run the installer, make sure that the **Desktop development with C++** workload is checked. Don't worry if you didn't install this workload when you installed Visual Studio. You can run the installer again and install it now.

  
  Desktop development with C++
  Build classic Windows-based applications using the power of the Visual C++ toolset, ATL, and optional features like...

- An understanding of the basics of using the Visual Studio IDE. If you've used Windows desktop apps before, you can probably keep up. For an introduction, see Visual Studio IDE feature tour.

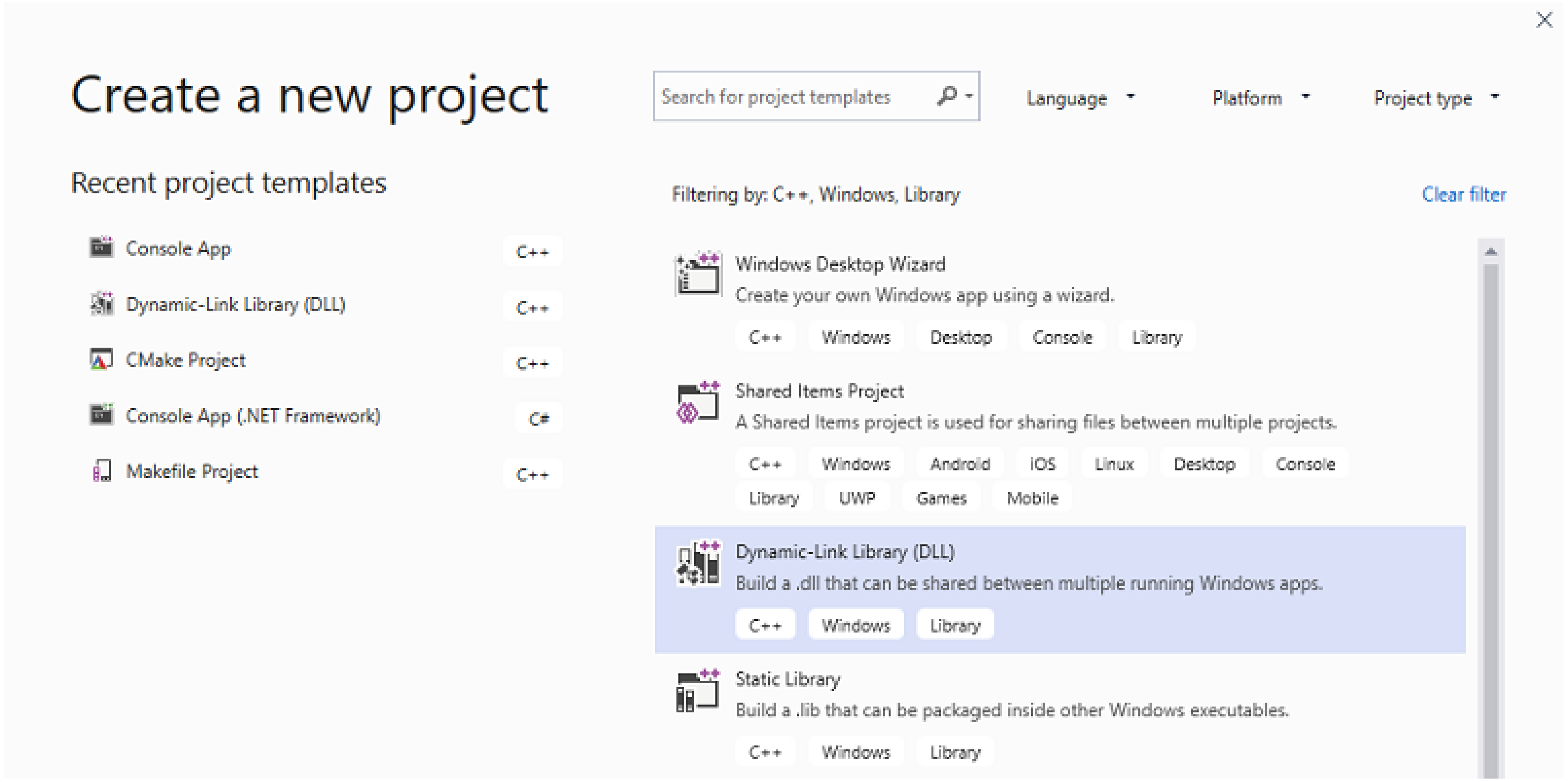
- An understanding of enough of the fundamentals of the C++ language to follow along. Don't worry, we don't do anything too complicated.

Create the DLL project

In this set of tasks, you create a project for your DLL, add code, and build it. To begin, start the Visual Studio IDE, and sign in if you need to. The instructions vary slightly depending on which version of Visual Studio you're using. Make sure you have the correct version selected in the control in the upper left of this page.

To create a DLL project in Visual Studio 2019

1. On the menu bar, choose **File** > **New** > **Project** to open the **Create a New Project** dialog box.


Create a new project

Search for project templates

Language ▾ Platform ▾ Project type ▾

Recent project templates

Console App	C++
Dynamic-Link Library (DLL)	C++
CMake Project	C++
Console App (.NET Framework)	C#
Makefile Project	C++

Filtering by: C++, Windows, Library

Clear filter

Windows Desktop Wizard
Create your own Windows app using a wizard.
C++ Windows Desktop Console Library

Shared Items Project
A Shared Items project is used for sharing files between multiple projects.
C++ Windows Android iOS Linux Desktop Console Library UWP Games Mobile

Dynamic-Link Library (DLL)
Build a .dll that can be shared between multiple running Windows apps.
C++ Windows Library

Static Library
Build a .lib that can be packaged inside other Windows executables.
C++ Windows Library

2. At the top of the dialog, set **Language** to **C++**, set **Platform** to **Windows**, and set **Project type** to **Library**.

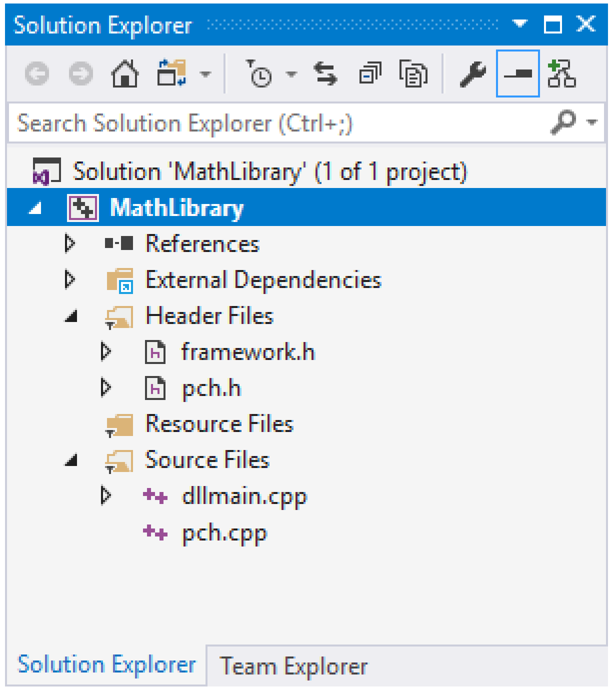
3. From the filtered list of project types, select **Dynamic-link Library (DLL)**, and then choose **Next**.

4. In the **Configure your new project** page, enter *MathLibrary* in the **Project name** box to specify a name for the project. Leave the default **Location** and **Solution name** values. Set **Solution** to **Create new solution**. Uncheck **Place solution and project in the same directory** if it's checked.

5. Choose the **Create** button to create the project.

5. Choose the **Create** button to create the project.

When the solution is created, you can see the generated project and source files in the **Solution Explorer** window in Visual Studio.

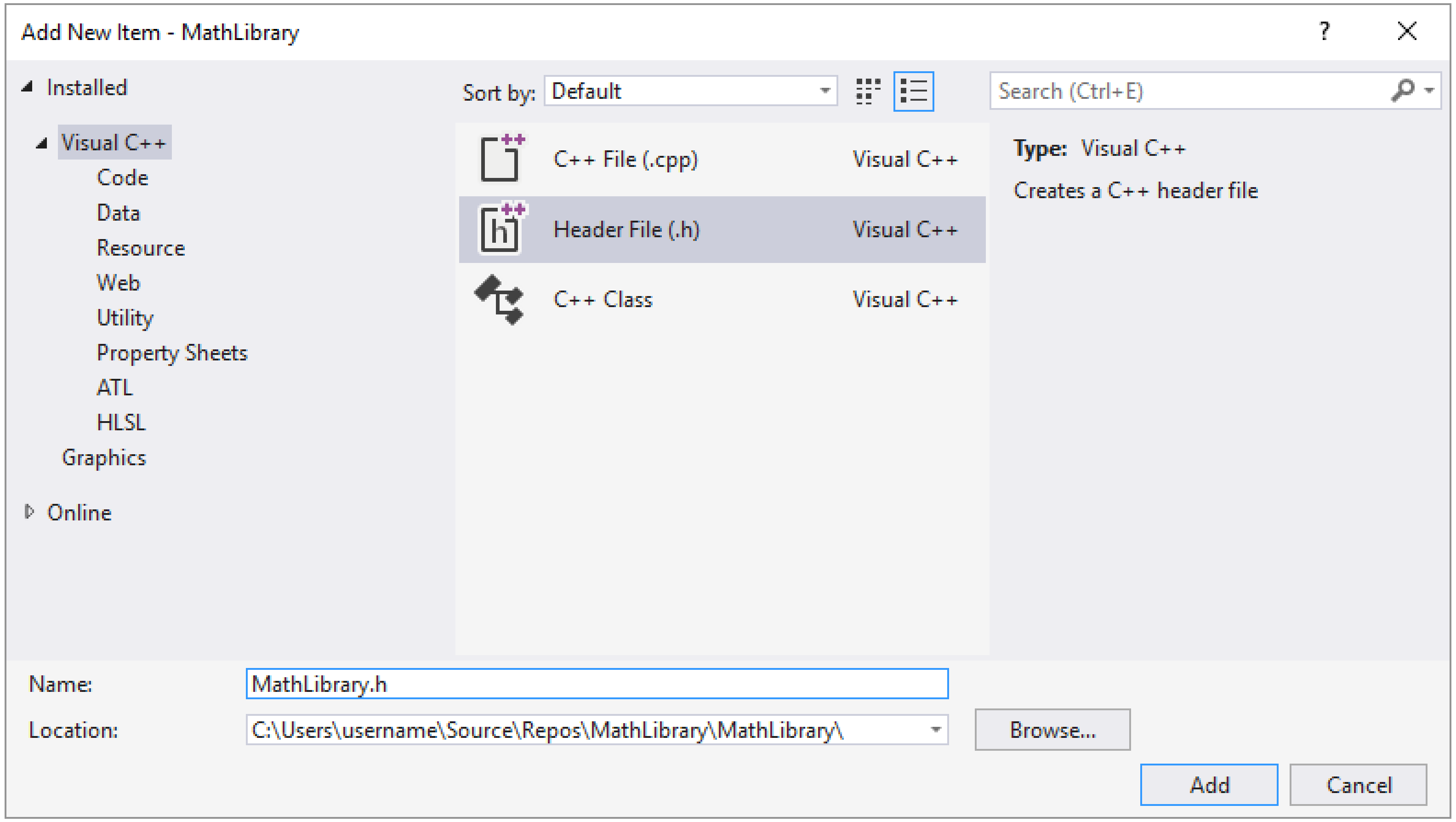


Right now, this DLL doesn't do very much. Next, you'll create a header file to declare the functions your DLL exports, and then add the function definitions to the DLL to make it more useful.

To add a header file to the DLL

1. To create a header file for your functions, on the menu bar, choose **Project** > **Add New Item**.

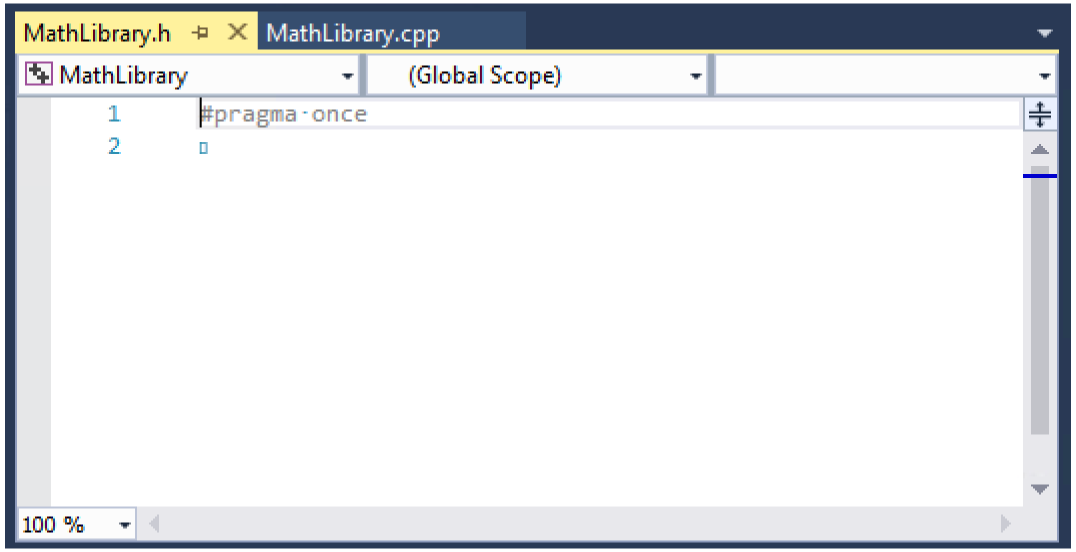
2. In the **Add New Item** dialog box, in the left pane, select **Visual C++**. In the center pane, select **Header File (.h)**. Specify *MathLibrary.h* as the name for the header file.



Name: MathLibrary.h

Location: C:\Users\username\Source\Repos\MathLibrary\MathLibrary\

3. Choose the **Add** button to generate a blank header file, which is displayed in a new editor window.



```
#pragma once
```

4. Replace the contents of the header file with this code:

C++ Copy

```cpp
// MathLibrary.h - Contains declarations of math functions
#pragma once

#ifdef MATHLIBRARY_EXPORTS
#define MATHLIBRARY_API __declspec(dllexport)
#else
#define MATHLIBRARY_API __declspec(dllimport)
#endif

// The Fibonacci recurrence relation describes a sequence F
// where F(n) is { n = 0, a
//               { n = 1, b
//               { n > 1, F(n-2) + F(n-1)
//                 { n > 1, F(n-2) + F(n-1)
// for some initial integral values a and b.
// If the sequence is initialized F(0) = 1, F(1) = 1,
// then this relation produces the well-known Fibonacci
// sequence: 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

// Initialize a Fibonacci relation sequence
// such that F(0) = a, F(1) = b.
// This function must be called before any other function.
extern "C" MATHLIBRARY_API void fibonacci_init(
    const unsigned long long a, const unsigned long long b);

// Produce the next value in the sequence.
// Returns true on success and updates current value and index;
// false on overflow, leaves current value and index unchanged.
extern "C" MATHLIBRARY_API bool fibonacci_next();

// Get the current value in the sequence.
extern "C" MATHLIBRARY_API unsigned long long fibonacci_current();

// Get the position of the current value in the sequence.
extern "C" MATHLIBRARY_API unsigned fibonacci_index();
```

This header file declares some functions to produce a generalized Fibonacci sequence, given two initial values. A call to `fibonacci_init(1, 1)` generates the familiar Fibonacci number sequence.

Notice the preprocessor statements at the top of the file. The new project template for a DLL project adds ***PROJECTNAME*_EXPORTS** to the defined preprocessor macros. In this example, Visual Studio defines **MATHLIBRARY_EXPORTS** when your MathLibrary DLL project is built.

When the **MATHLIBRARY_EXPORTS** macro is defined, the **MATHLIBRARY_API** macro sets the `__declspec(dllexport)` modifier on the function declarations. This modifier tells the compiler and linker to export a function or variable from the DLL for use by other applications. When **MATHLIBRARY_EXPORTS** is undefined, for example, when the header file is included by a client application, **MATHLIBRARY_API** applies the `__declspec(dllimport)` modifier to the declarations. This modifier optimizes the import of the function or variable in an application. For more information, see dllexport, dllimport.

To add an implementation to the DLL

1. In **Solution Explorer**, right-click on the **Source Files** node and choose **Add** > **New Item**. Create a new .cpp file called *MathLibrary.cpp*, in the same way that you added a new header file in the previous step.

2. In the editor window, select the tab for **MathLibrary.cpp** if it's already open. If not, in **Solution Explorer**, double-click **MathLibrary.cpp** in the **Source Files** folder of the **MathLibrary** project to open it.

3. In the editor, replace the contents of the MathLibrary.cpp file with the following code:

C++

```cpp
// MathLibrary.cpp : Defines the exported functions for the DLL.
#include "pch.h" // use stdafx.h in Visual Studio 2017 and earlier
#include <utility>
#include <limits.h>
#include "MathLibrary.h"

// DLL internal state variables:
static unsigned long long previous_;  // Previous value, if any
static unsigned long long current_;   // Current sequence value
static unsigned index_;               // Current seq. position

// Initialize a Fibonacci relation sequence
// such that F(0) = a, F(1) = b.
// This function must be called before any other function.
void fibonacci_init(
    const unsigned long long a,
    const unsigned long long b)
{
    index_ = 0;
    current_ = a;
    previous_ = b; // see special case when initialized
}

// Produce the next value in the sequence.
// Returns true on success, false on overflow.
bool fibonacci_next()
{
    // check to see if we'd overflow result or position
    if ((ULLONG_MAX - previous_ < current_) ||
        (UINT_MAX == index_))
    {
        return false;
    }

    // Special case when index == 0, just return b value
    if (index_ > 0)
    {
        // otherwise, calculate next sequence value
        previous_ += current_;
    }
    std::swap(current_, previous_);
    ++index_;
    return true;
}

// Get the current value in the sequence.
unsigned long long fibonacci_current()
{
    return current_;
}

// Get the current index position in the sequence.
unsigned fibonacci_index()
{
    return index_;
}
```

To verify that everything works so far, compile the dynamic link library. To compile, choose **Build** > **Build Solution** on the menu bar. The DLL and related compiler output are placed in a folder called *Debug* directly below the solution folder. If you create a Release build, the output is placed in a folder called *Release*. The output should look something like this:

Output

```
1>------ Build started: Project: MathLibrary, Configuration: Debug Win32 ------
1>pch.cpp
1>dllmain.cpp
1>MathLibrary.cpp
1>Generating Code...
1>   Creating library C:\Users\username\Source\Repos\MathLibrary\Debug\MathLibrary.lib and object
C:\Users\username\Source\Repos\MathLibrary\Debug\MathLibrary.exp
1>MathLibrary.vcxproj -> C:\Users\username\Source\Repos\MathLibrary\Debug\MathLibrary.dll
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Congratulations, you've created a DLL using Visual Studio! Next, you'll create a client app that uses the functions exported by the DLL.

Create a client app that uses the DLL

When you create a DLL, think about how client apps may use it. To call the functions or access the data exported by a DLL, client source code must have the declarations available at compile time. At link time, the linker requires information to resolve the function calls or data accesses. A DLL supplies this information in an *import library*, a file that contains information about how to find the functions and data, instead of the actual code. And at run time, the DLL must be available to the client, in a location that the operating system can find.

Whether it's your own or from a third-party, your client app project needs several pieces of information to use a DLL. It needs to find the headers that declare the DLL exports, the import libraries for the linker, and the DLL itself. One solution is to copy all of these files into your client project. For third-party DLLs that are unlikely to change while your client is in development, this method may be the best way to use them. However, when you also build the DLL, it's better to avoid duplication. If you make a local copy of DLL files that are under development, you may accidentally change a header file in one copy but not the other, or use an out-of-date library.

To avoid out-of-sync code, we recommend you set the include path in your client project to include the DLL header files directly from your DLL project. Also, set the library path in your client project to include the DLL import libraries from the DLL project. And finally, copy the built DLL from the DLL project into your client build output directory. This step allows your client app to use the same DLL code you build.

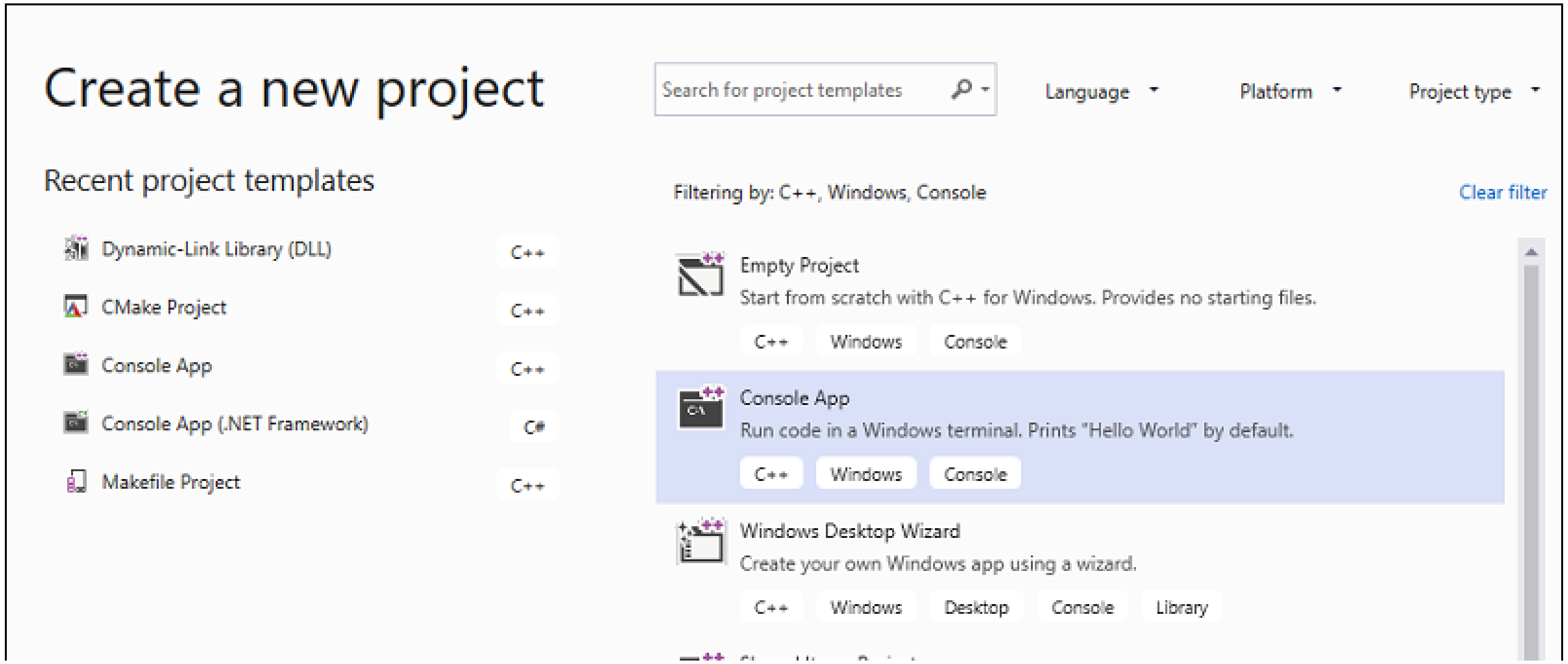
To create a client app in Visual Studio

1. On the menu bar, choose **File** > **New** > **Project** to open the **Create a new project** dialog box.

2. At the top of the dialog, set **Language** to **C++**, set **Platform** to **Windows**, and set **Project type** to **Console**.

3. From the filtered list of project types, choose **Console App** then choose **Next**.

4. In the **Configure your new project** page, enter *MathClient* in the **Project name** box to specify a name for the project. Leave the default **Location** and **Solution name** values. Set **Solution** to **Create new solution**. Uncheck **Place solution and project in the same directory** if it's checked.



Create a new project

Search for project templates

Language
Platform
Project type

Recent project templates

Dynamic-Link Library (DLL)	C++
CMake Project	C++
Console App	C++
Console App (.NET Framework)	C#
Makefile Project	C++

Filtering by: C++, Windows, Console

Clear filter

Empty Project
Start from scratch with C++ for Windows. Provides no starting files.
C++ Windows Console

Console App
Run code in a Windows terminal. Prints "Hello World" by default.
C++ Windows Console

Windows Desktop Wizard
Create your own Windows app using a wizard.
C++ Windows Desktop Console Library

5. Choose the **Create** button to create the client project.

A minimal console application project is created for you. The name for the main source file is the same as the project name that you entered earlier. In this example, it's named **MathClient.cpp**. You can build it, but it doesn't use your DLL yet.

Next, to call the MathLibrary functions in your source code, your project must include the *MathLibrary.h* file. You could copy this header file into your client app project, then add it to the project as an existing item. This method can be a good choice for third-party libraries. However, if you're working on the code for your DLL and your client at the same time, the header files could get out of sync. To avoid this issue, set the **Additional Include Directories** path in your project to include the path to the original header.

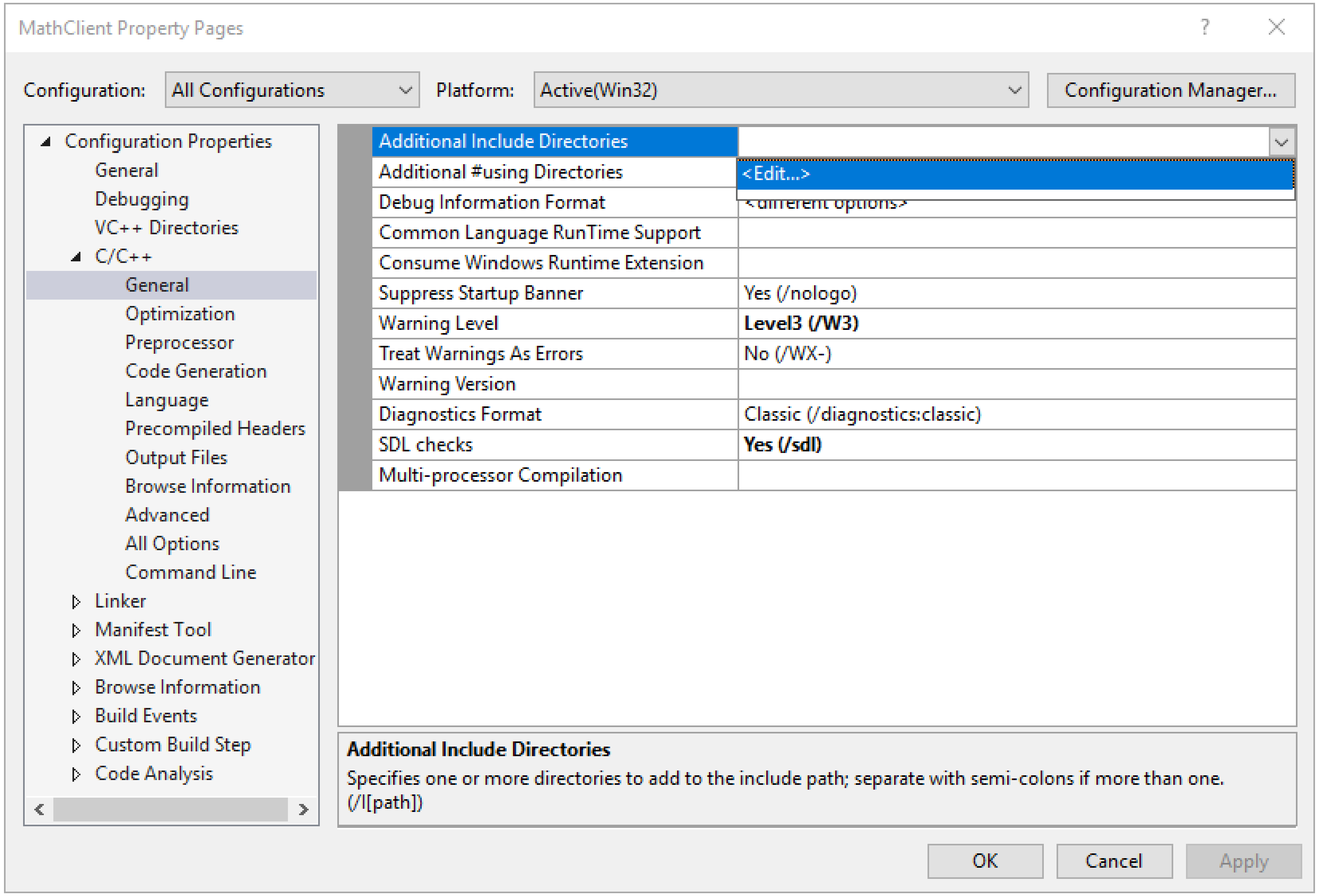
To add the DLL header to your include path

1. Right-click on the **MathClient** node in **Solution Explorer** to open the **Property Pages** dialog.

2. In the **Configuration** drop-down box, select **All Configurations** if it's not already selected.

3. In the left pane, select **Configuration Properties** > **C/C++** > **General**.

4. In the property pane, select the drop-down control next to the **Additional Include Directories** edit box, and then choose **Edit**.


MathClient Property Pages

Configuration: All Configurations   Platform: Active(Win32)   Configuration Manager...

- Configuration Properties
  - General
  - Debugging
  - VC++ Directories
  - C/C++
    - General
    - Optimization
    - Preprocessor
    - Code Generation
    - Language
    - Precompiled Headers
    - Output Files
    - Browse Information
    - Advanced
    - All Options
    - Command Line
  - Linker
  - Manifest Tool
  - XML Document Generator
  - Browse Information
  - Build Events
  - Custom Build Step
  - Code Analysis

Additional Include Directories	
Additional #using Directories	<Edit...>
Debug Information Format	<different options>
Common Language RunTime Support	
Consume Windows Runtime Extension	
Suppress Startup Banner	Yes (/nologo)
Warning Level	**Level3 (/W3)**
Treat Warnings As Errors	No (/WX-)
Warning Version	
Diagnostics Format	Classic (/diagnostics:classic)
SDL checks	**Yes (/sdl)**
Multi-processor Compilation	

Additional Include Directories
Specifies one or more directories to add to the include path; separate with semi-colons if more than one. (/I[path])

OK   Cancel   Apply

5. Double-click in the top pane of the **Additional Include Directories** dialog box to enable an edit control. Or, choose the folder

icon to create a new entry.

6. In the edit control, specify the path to the location of the **MathLibrary.h** header file. You can choose the ellipsis (**...**) control to browse to the correct folder.

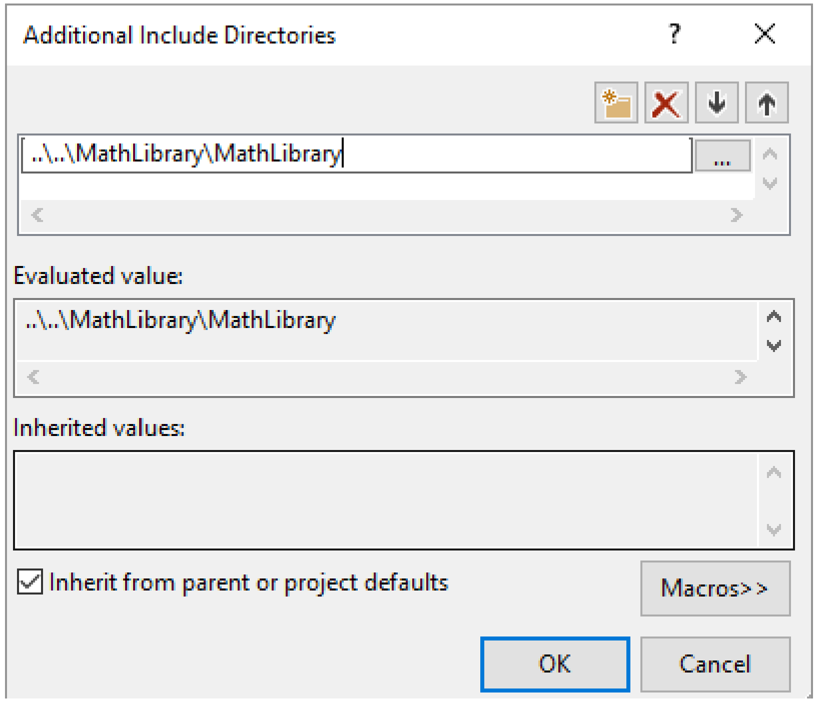
   You can also enter a relative path from your client source files to the folder that contains the DLL header files. If you followed the directions to put your client project in a separate solution from the DLL, the relative path should look like this:

   `..\..\MathLibrary\MathLibrary`

   If your DLL and client projects are in the same solution, the relative path might look like this:

   `..\MathLibrary`

   When the DLL and client projects are in other folders, adjust the relative path to match. Or, use the ellipsis control to browse for the folder.



..\..\MathLibrary\MathLibrary

Evaluated value:

..\..\MathLibrary\MathLibrary

Inherited values:

☑ Inherit from parent or project defaults