Contents

Install C and C++ support in Visual Studio

9/2/2022 • 14 minutes to read • Edit Online

If you haven't downloaded and installed Visual Studio and the Microsoft C/C++ tools yet, here's how to get started.

Visual Studio 2022 Installation

Welcome to Visual Studio 2022! In this version, it's easy to choose and install just the features you need. And because of its reduced minimum footprint, it installs quickly and with less system impact.

> **NOTE**
>
> This topic applies to installation of Visual Studio on Windows. Visual Studio Code is a lightweight, cross-platform development environment that runs on Windows, Mac, and Linux systems. The Microsoft C/C++ for Visual Studio Code extension supports IntelliSense, debugging, code formatting, auto-completion. Visual Studio for Mac doesn't support Microsoft C++, but does support .NET languages and cross-platform development. For installation instructions, see Install Visual Studio for Mac.

Want to know more about what else is new in this version? See the Visual Studio release notes.

Ready to install? We'll walk you through it, step-by-step.

Step 1 - Make sure your computer is ready for Visual Studio

Before you begin installing Visual Studio:

1. Check the system requirements. These requirements help you know whether your computer supports Visual Studio 2022.
2. Apply the latest Windows updates. These updates ensure that your computer has both the latest security updates and the required system components for Visual Studio.
3. Reboot. The reboot ensures that any pending installs or updates don't hinder the Visual Studio install.
4. Free up space. Remove unneeded files and applications from your %SystemDrive% by, for example, running the Disk Cleanup app.

For questions about running previous versions of Visual Studio side by side with Visual Studio 2022, see the Visual Studio 2022 Platform Targeting and Compatibility page.

Step 2 - Download Visual Studio

Next, download the Visual Studio bootstrapper file. To do so, choose the following button to go to the Visual Studio download page. Select the edition of Visual Studio that you want and choose the **Free trial** or **Free download** button.

DOWNLOAD VISUAL STUDIO

Step 3 - Install the Visual Studio installer

Run the bootstrapper file you downloaded to install the Visual Studio Installer. This new lightweight installer includes everything you need to both install and customize Visual Studio.

1. From your **Downloads** folder, double-click the bootstrapper that matches or is similar to one of the

following files:

- **vs_community.exe** for Visual Studio Community
- **vs_professional.exe** for Visual Studio Professional
- **vs_enterprise.exe** for Visual Studio Enterprise

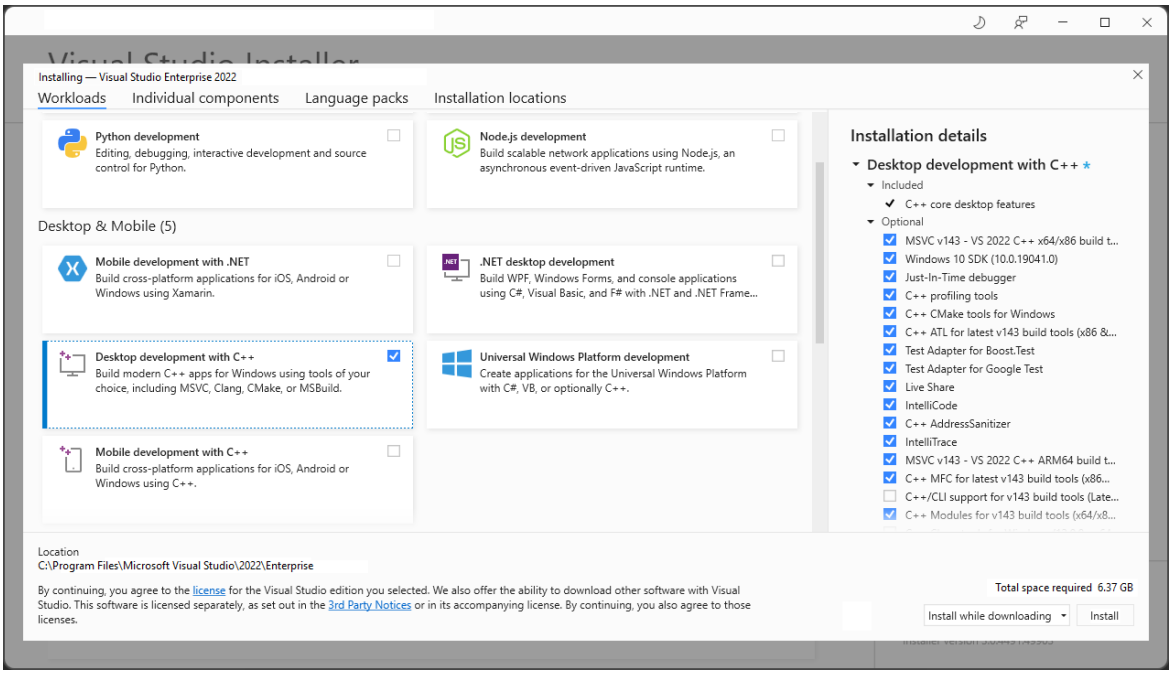
If you receive a User Account Control notice, choose **Yes** to allow the bootstrapper to run.

2. We'll ask you to acknowledge the Microsoft License Terms and the Microsoft Privacy Statement. Choose **Continue**.

Step 4 - Choose workloads

After the installer is installed, you can use it to customize your installation by selecting the *workloads*, or feature sets, that you want. Here's how.

1. Find the workload you want in the **Installing Visual Studio** screen.



For core C and C++ support, choose the "Desktop development with C++" workload. It comes with the default core editor, which includes basic code editing support for over 20 languages, the ability to open and edit code from any folder without requiring a project, and integrated source code control.

Additional workloads support other kinds of development. For example, choose the "Universal Windows Platform development" workload to create apps that use the Windows Runtime for the Microsoft Store. Choose "Game development with C++" to create games that use DirectX, Unreal, and Cocos2d. Choose "Linux development with C++" to target Linux platforms, including IoT development.

The **Installation details** pane lists the included and optional components installed by each workload. You can select or deselect optional components in this list. For example, to support development by using the Visual Studio 2017 or 2015 compiler toolsets, choose the MSVC v141 or MSVC v140 optional components. You can add support for MFC, the experimental Modules language extension, IncrediBuild, and more.

2. After you choose the workload(s) and optional components you want, choose **Install**.

   Next, status screens appear that show the progress of your Visual Studio installation.

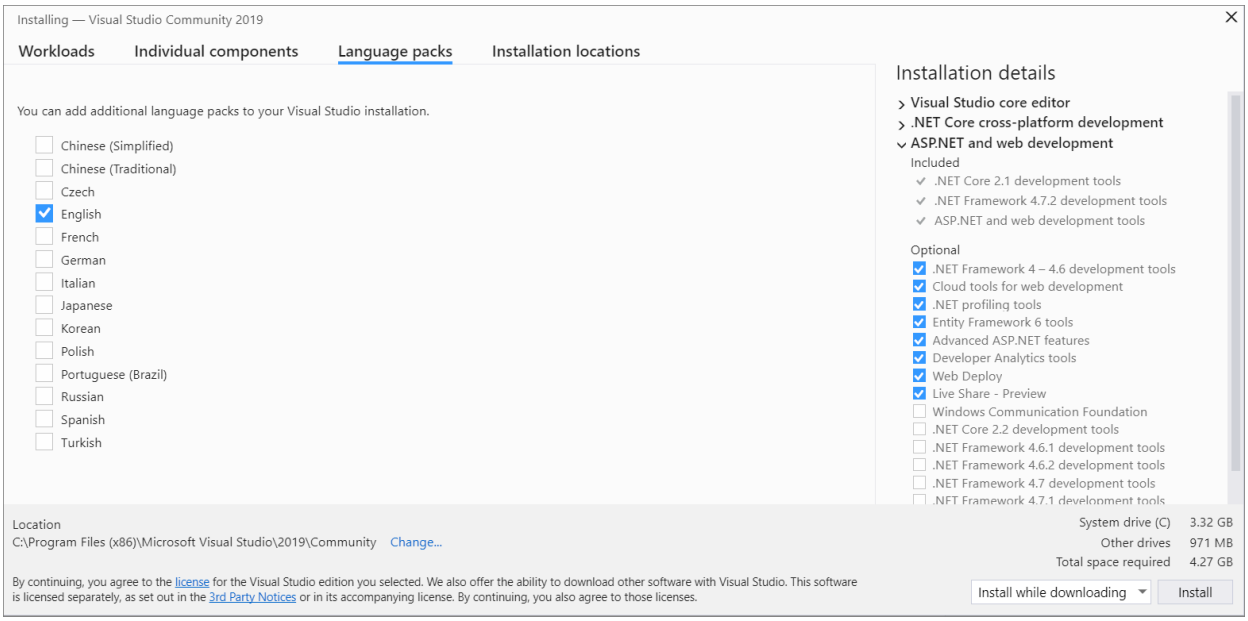
> **TIP**
>
> At any time after installation, you can install workloads or components that you didn't install initially. If you have Visual Studio open, go to **Tools** > **Get Tools and Features...** which opens the Visual Studio Installer. Or, open **Visual Studio Installer** from the Start menu. From there, you can choose the workloads or components that you wish to install. Then, choose **Modify**.

Step 5 - Choose individual components (Optional)

If you don't want to use the Workloads feature to customize your Visual Studio installation, or you want to add more components than a workload installs, you can do so by installing or adding individual components from the **Individual components** tab. Choose what you want, and then follow the prompts.

Step 6 - Install language packs (Optional)

By default, the installer program tries to match the language of the operating system when it runs for the first time. To install Visual Studio in a language of your choosing, choose the **Language packs** tab from the Visual Studio Installer, and then follow the prompts.



Change the installer language from the command line

Another way that you can change the default language is by running the installer from the command line. For example, you can force the installer to run in English by using the following command: `vs_installer.exe --locale en-US`. The installer will remember this setting when it's run the next time. The installer supports the following language tokens: zh-cn, zh-tw, cs-cz, en-us, es-es, fr-fr, de-de, it-it, ja-jp, ko-kr, pl-pl, pt-br, ru-ru, and tr-tr.

Step 7 - Change the installation location (Optional)

You can reduce the installation footprint of Visual Studio on your system drive. You can choose to move the download cache, shared components, SDKs, and tools to different drives, and keep Visual Studio on the drive that runs it the fastest.

> **IMPORTANT**
>
> You can select a different drive only when you first install Visual Studio. If you've already installed it and want to change drives, you must uninstall Visual Studio and then reinstall it.

Step 8 - Start developing

1. After Visual Studio installation is complete, choose the **Launch** button to get started developing with

Visual Studio.

2. On the start window, choose **Create a new project**.

3. In the search box, enter the type of app you want to create to see a list of available templates. The list of templates depends on the workload(s) that you chose during installation. To see different templates, choose different workloads.

   You can also filter your search for a specific programming language by using the **Language** drop-down list. You can filter by using the **Platform** list and the **Project type** list, too.

4. Visual Studio opens your new project, and you're ready to code!

Visual Studio 2019 Installation

Welcome to Visual Studio 2019! In this version, it's easy to choose and install just the features you need. And because of its reduced minimum footprint, it installs quickly and with less system impact.

> **NOTE**
> This topic applies to installation of Visual Studio on Windows. Visual Studio Code is a lightweight, cross-platform development environment that runs on Windows, Mac, and Linux systems. The Microsoft C/C++ for Visual Studio Code extension supports IntelliSense, debugging, code formatting, auto-completion. Visual Studio for Mac doesn't support Microsoft C++, but does support .NET languages and cross-platform development. For installation instructions, see Install Visual Studio for Mac.

Want to know more about what else is new in this version? See the Visual Studio release notes.

Ready to install? We'll walk you through it, step-by-step.

Step 1 - Make sure your computer is ready for Visual Studio

Before you begin installing Visual Studio:

1. Check the system requirements. These requirements help you know whether your computer supports Visual Studio 2019.

2. Apply the latest Windows updates. These updates ensure that your computer has both the latest security updates and the required system components for Visual Studio.

3. Reboot. The reboot ensures that any pending installs or updates don't hinder the Visual Studio install.

4. Free up space. Remove unneeded files and applications from your %SystemDrive% by, for example, running the Disk Cleanup app.

For questions about running previous versions of Visual Studio side by side with Visual Studio 2019, see the Visual Studio 2019 Platform Targeting and Compatibility page.

Step 2 - Download Visual Studio

Next, download the Visual Studio bootstrapper file. To do so, choose the following button to go to the Visual Studio download page. Choose the Download button, then you can select the edition of Visual Studio that you want.

DOWNLOAD VISUAL STUDIO 2019

Step 3 - Install the Visual Studio installer

Run the bootstrapper file you downloaded to install the Visual Studio Installer. This new lightweight installer includes everything you need to both install and customize Visual Studio.

1. From your **Downloads** folder, double-click the bootstrapper that matches or is similar to one of the following files:

   - **vs_community.exe** for Visual Studio Community
   - **vs_professional.exe** for Visual Studio Professional
   - **vs_enterprise.exe** for Visual Studio Enterprise

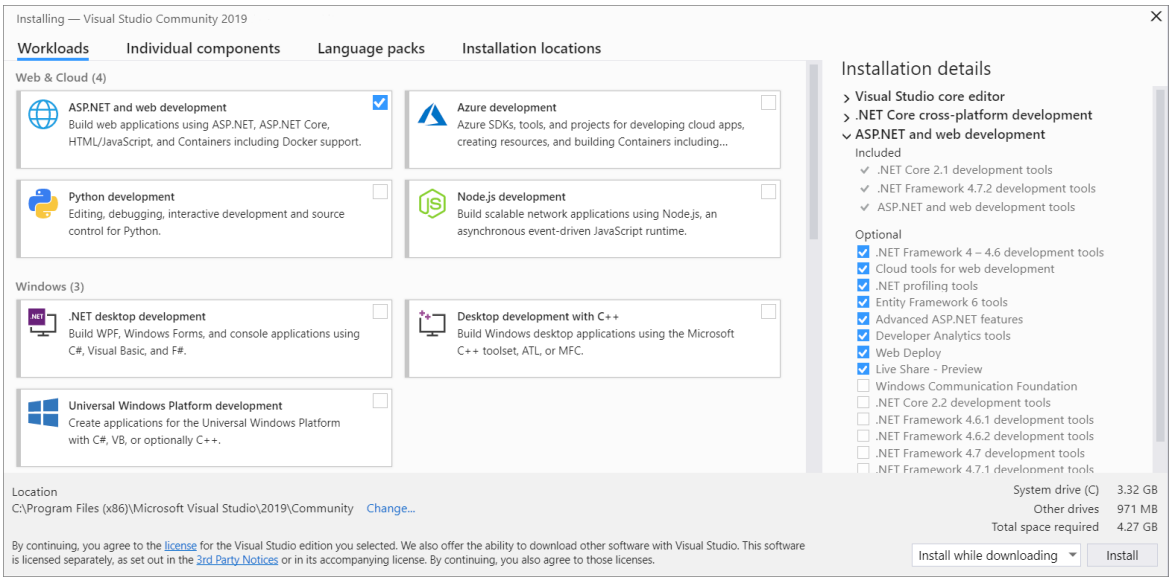
   If you receive a User Account Control notice, choose **Yes** to allow the bootstrapper to run.

2. We'll ask you to acknowledge the Microsoft License Terms and the Microsoft Privacy Statement. Choose **Continue**.

Step 4 - Choose workloads

After the installer is installed, you can use it to customize your installation by selecting the *workloads*, or feature sets, that you want. Here's how.

1. Find the workload you want in the **Installing Visual Studio** screen.



   For core C and C++ support, choose the "Desktop development with C++" workload. It comes with the default core editor, which includes basic code editing support for over 20 languages, the ability to open and edit code from any folder without requiring a project, and integrated source code control.

   Additional workloads support other kinds of development. For example, choose the "Universal Windows Platform development" workload to create apps that use the Windows Runtime for the Microsoft Store. Choose "Game development with C++" to create games that use DirectX, Unreal, and Cocos2d. Choose "Linux development with C++" to target Linux platforms, including IoT development.

   The **Installation details** pane lists the included and optional components installed by each workload. You can select or deselect optional components in this list. For example, to support development by using the Visual Studio 2017 or 2015 compiler toolsets, choose the MSVC v141 or MSVC v140 optional components. You can add support for MFC, the experimental Modules language extension, IncrediBuild, and more.

2. After you choose the workload(s) and optional components you want, choose **Install**.

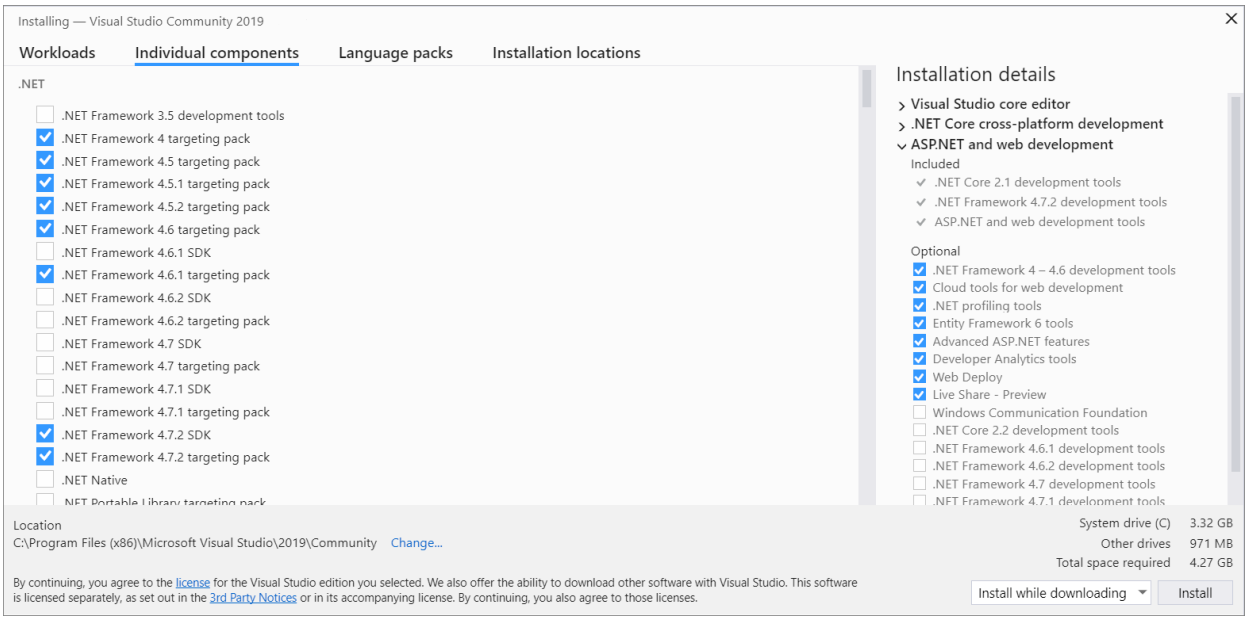
   Next, status screens appear that show the progress of your Visual Studio installation.

TIP

At any time after installation, you can install workloads or components that you didn't install initially. If you have Visual Studio open, go to **Tools** > **Get Tools and Features...** which opens the Visual Studio Installer. Or, open **Visual Studio Installer** from the Start menu. From there, you can choose the workloads or components that you wish to install. Then, choose **Modify**.

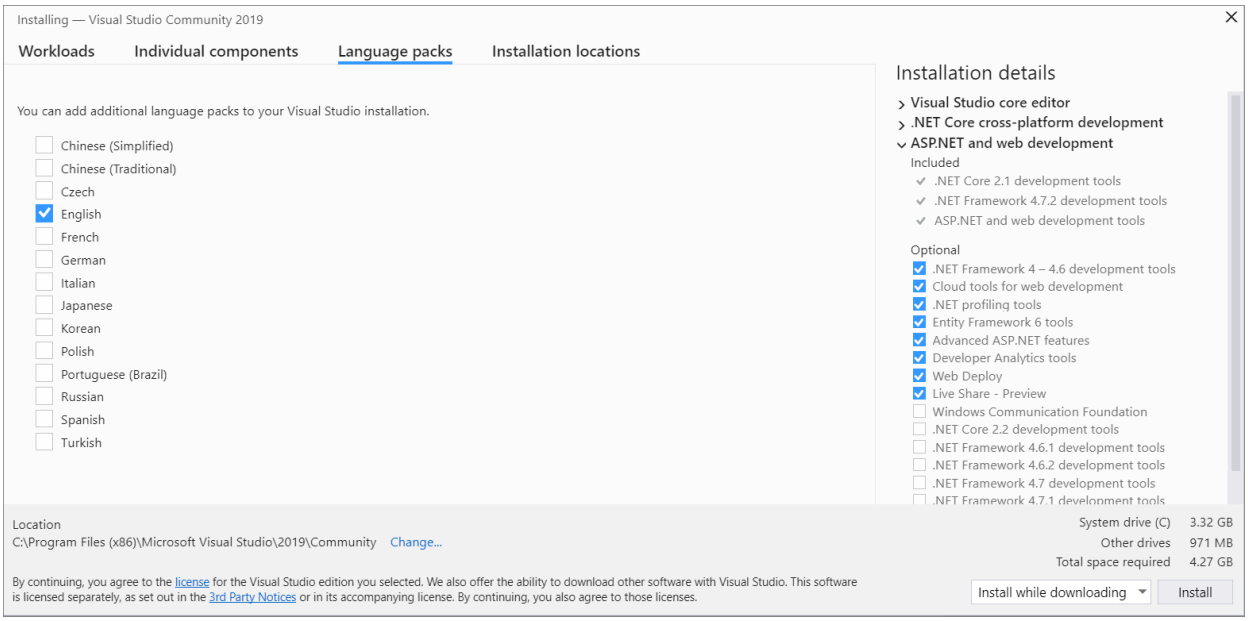
Step 5 - Choose individual components (Optional)

If you don't want to use the Workloads feature to customize your Visual Studio installation, or you want to add more components than a workload installs, you can do so by installing or adding individual components from the **Individual components** tab. Choose what you want, and then follow the prompts.



Step 6 - Install language packs (Optional)

By default, the installer program tries to match the language of the operating system when it runs for the first time. To install Visual Studio in a language of your choosing, choose the **Language packs** tab from the Visual Studio Installer, and then follow the prompts.



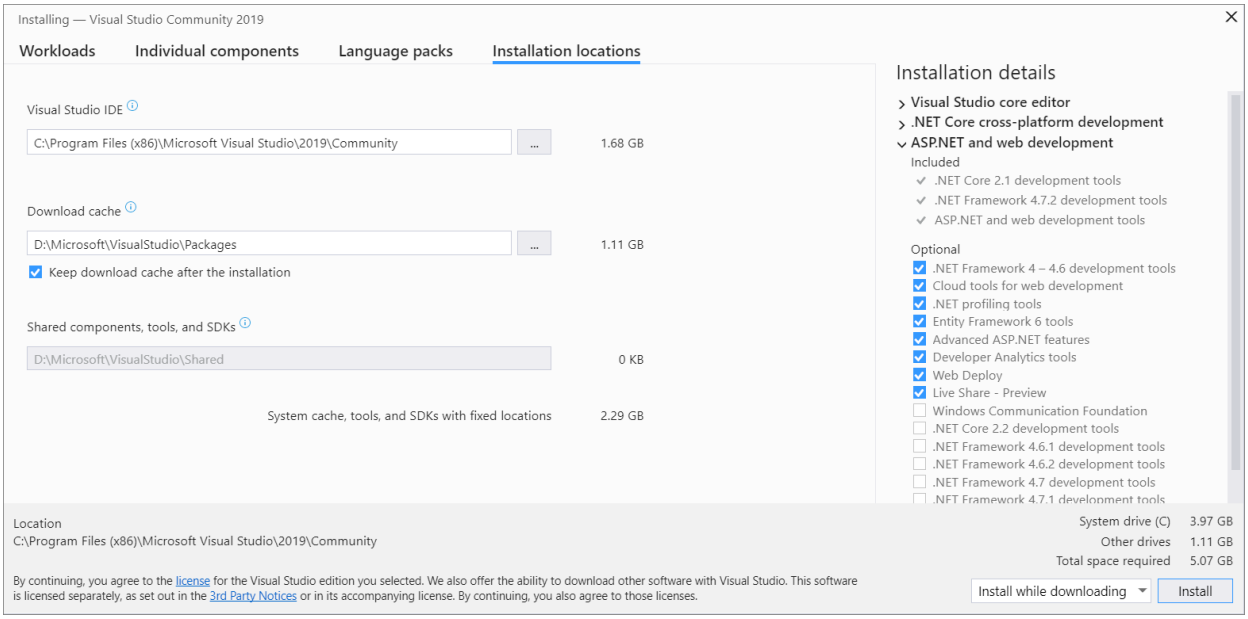
Change the installer language from the command line

Another way that you can change the default language is by running the installer from the command line. For example, you can force the installer to run in English by using the following command:

`vs_installer.exe --locale en-US`. The installer will remember this setting when it's run the next time. The installer supports the following language tokens: zh-cn, zh-tw, cs-cz, en-us, es-es, fr-fr, de-de, it-it, ja-jp, ko-kr, pl-pl, pt-br, ru-ru, and tr-tr.

Step 7 - Change the installation location (Optional)

You can reduce the installation footprint of Visual Studio on your system drive. You can choose to move the download cache, shared components, SDKs, and tools to different drives, and keep Visual Studio on the drive that runs it the fastest.



> **IMPORTANT**
>
> You can select a different drive only when you first install Visual Studio. If you've already installed it and want to change drives, you must uninstall Visual Studio and then reinstall it.

Step 8 - Start developing

1. After Visual Studio installation is complete, choose the **Launch** button to get started developing with Visual Studio.
2. On the start window, choose **Create a new project**.
3. In the search box, enter the type of app you want to create to see a list of available templates. The list of templates depends on the workload(s) that you chose during installation. To see different templates, choose different workloads.

   You can also filter your search for a specific programming language by using the **Language** drop-down list. You can filter by using the **Platform** list and the **Project type** list, too.
4. Visual Studio opens your new project, and you're ready to code!

Visual Studio 2017 Installation

In Visual Studio 2017, it's easy to choose and install just the features you need. And because of its reduced minimum footprint, it installs quickly and with less system impact.

Prerequisites

- A broadband internet connection. The Visual Studio installer can download several gigabytes of data.
- A computer that runs Microsoft Windows 7 or later versions. We recommend the latest version of Windows for the best development experience. Make sure that the latest updates are applied to your

system before you install Visual Studio.

- Enough free disk space. Visual Studio requires at least 7 GB of disk space, and can take 50 GB or more if many common options are installed. We recommend you install it on your C: drive.

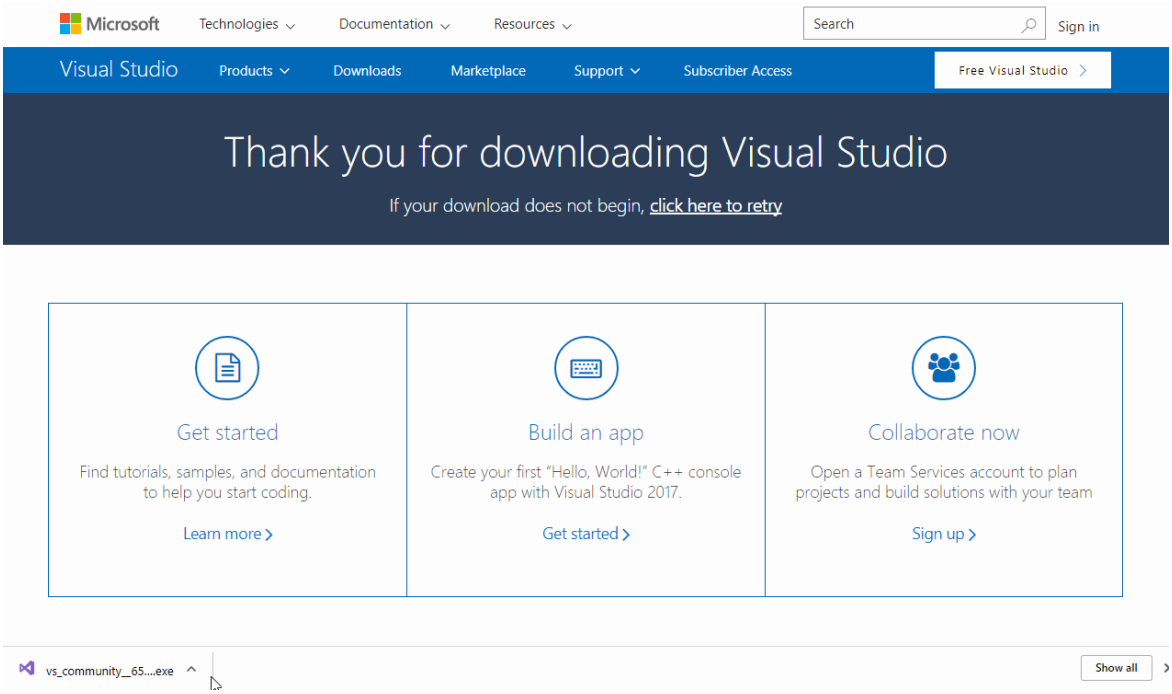
For details on the disk space and operating system requirements, see Visual Studio Product Family System Requirements. The installer reports how much disk space is required for the options you select.

Download and install

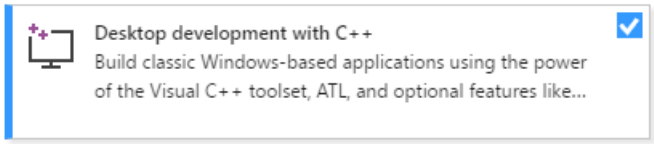
1. To download the latest Visual Studio 2017 installer for Windows, go to the Microsoft Visual Studio Older downloads page. Expand the **2017** section, and choose the **Download** button.

   > **TIP**
   > The Community edition is for individual developers, classroom learning, academic research, and open source development. For other uses, install Visual Studio 2017 Professional or Visual Studio 2017 Enterprise.

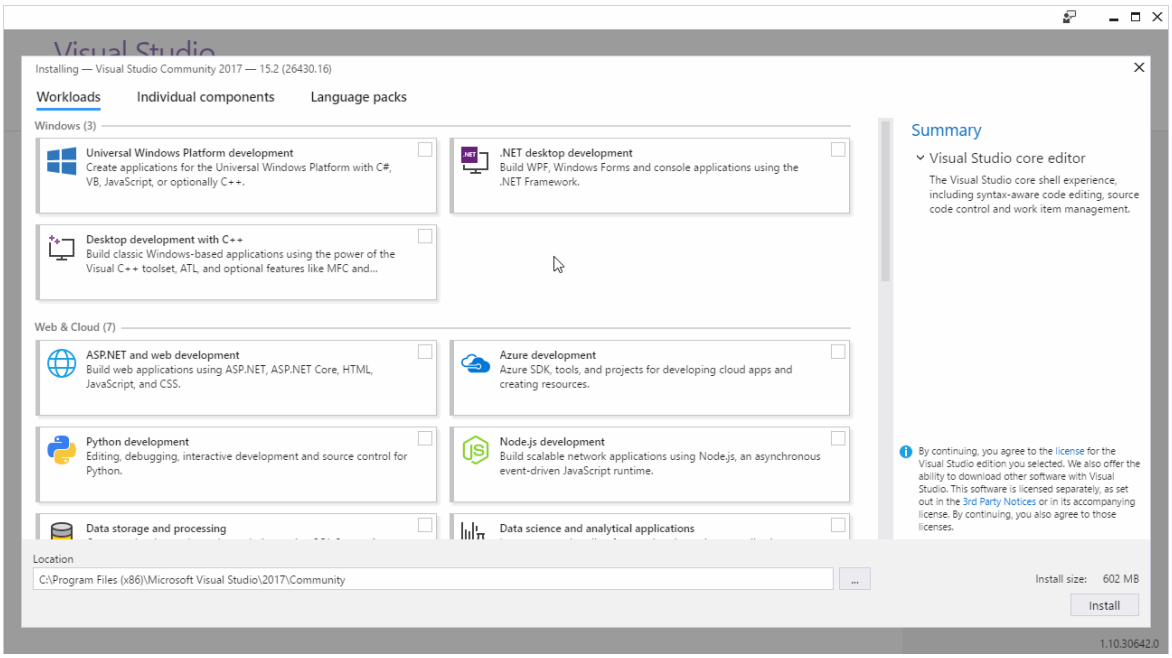
2. Find the installer file you downloaded and run it. The downloaded file may be displayed in your browser, or you may find it in your Downloads folder. The installer needs Administrator privileges to run. You may see a **User Account Control** dialog asking you to give permission to let the installer make changes to your system; choose **Yes**. If you're having trouble, find the downloaded file in File Explorer, right-click on the installer icon, and choose **Run as Administrator** from the context menu.



3. The installer presents you with a list of workloads, which are groups of related options for specific development areas. Support for C++ is now part of optional workloads that aren't installed by default.



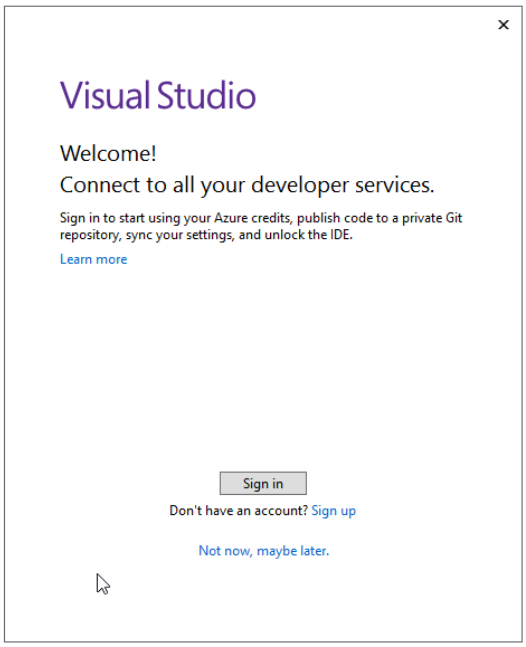
   For C and C++, select the **Desktop development with C++** workload and then choose **Install**.



4. When the installation completes, choose the **Launch** button to start Visual Studio.

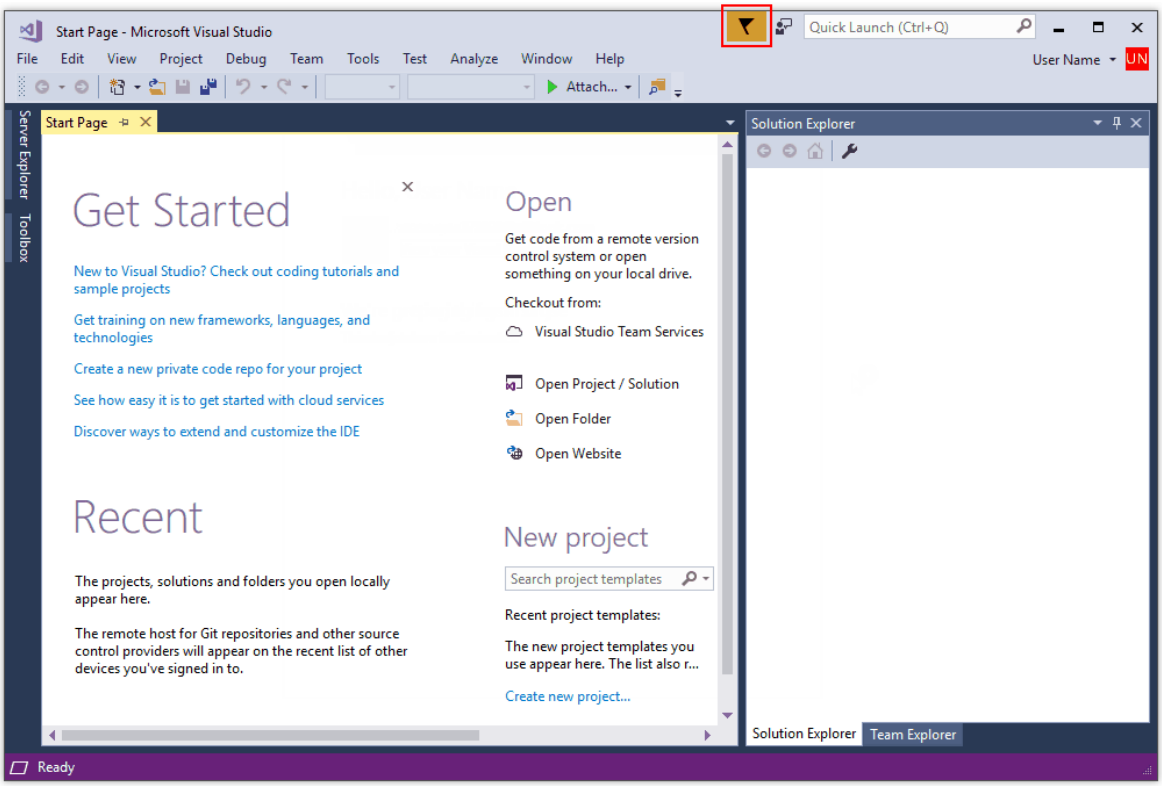
   The first time you run Visual Studio, you're asked to sign in with a Microsoft Account. If you don't have one, you can create one for free. You must also choose a theme. Don't worry, you can change it later if you want to.

   It may take Visual Studio several minutes to get ready for use the first time you run it. Here's what it looks like in a quick time-lapse:



Visual Studio starts much faster when you run it again.

5. When Visual Studio opens, check to see if the flag icon in the title bar is highlighted:



If it's highlighted, select it to open the **Notifications** window. If there are any updates available for Visual Studio, we recommend you install them now. Once the installation is complete, restart Visual Studio.

Visual Studio 2015 Installation

To install Visual Studio 2015, go to the Microsoft Visual Studio Older downloads page. Expand the **2015** section, and choose the **Download** button. Run the downloaded setup program and choose **Custom installation** and then choose the C++ component. To add C and C++ support to an existing Visual Studio 2015 installation, click on the Windows Start button and type **Add Remove Programs**. Open the program from the results list and then find your Visual Studio 2015 installation in the list of installed programs. Double-click it, then choose **Modify** and select the Visual C++ components to install.

In general, we highly recommend that you use the latest version of Visual Studio even if you need to compile your code using the Visual Studio 2015 compiler. For more information, see Use native multi-targeting in Visual Studio to build old projects.

When Visual Studio is running, you're ready to continue to the next step.

Next Steps

Create a C++ project

Create a C++ console app project

9/2/2022 • 6 minutes to read • Edit Online

The usual starting point for a C++ programmer is a "Hello, world!" application that runs on the command line. That's what you'll create in Visual Studio in this step.

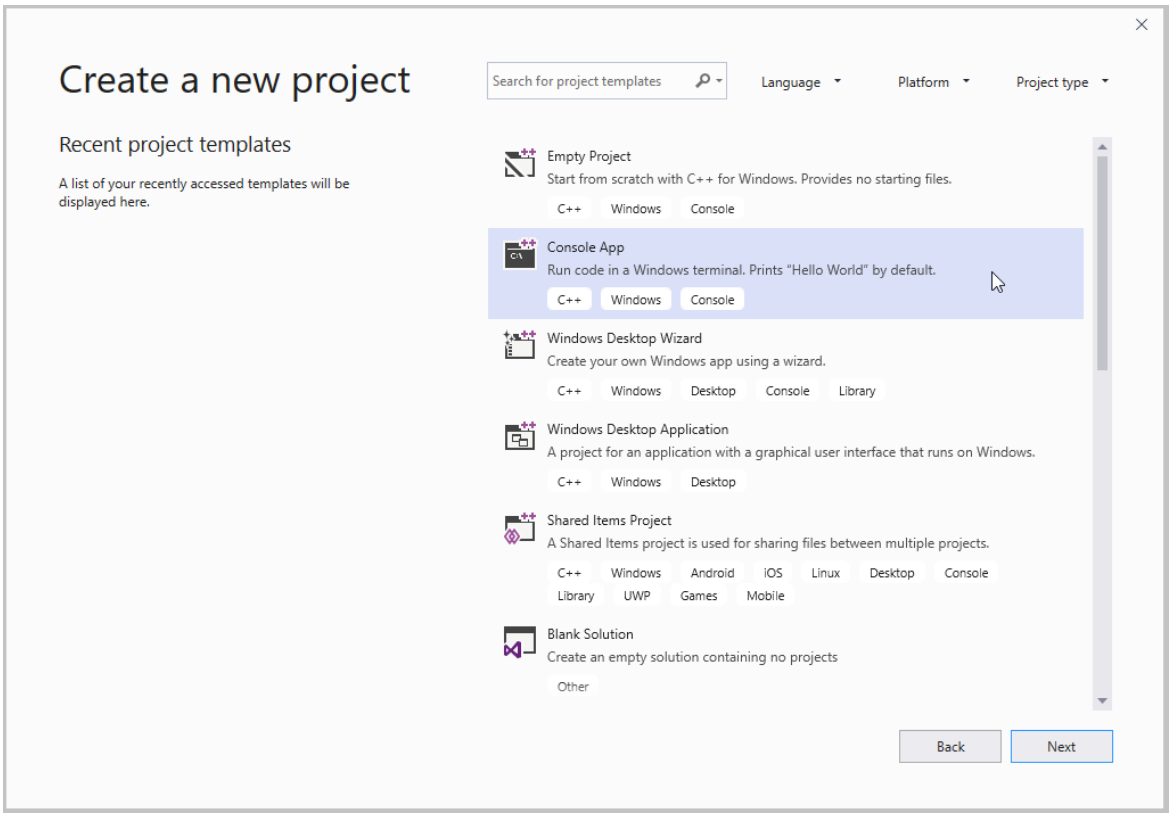
Prerequisites

- Have Visual Studio with the Desktop development with C++ workload installed and running on your computer. If it's not installed yet, see Install C++ support in Visual Studio.

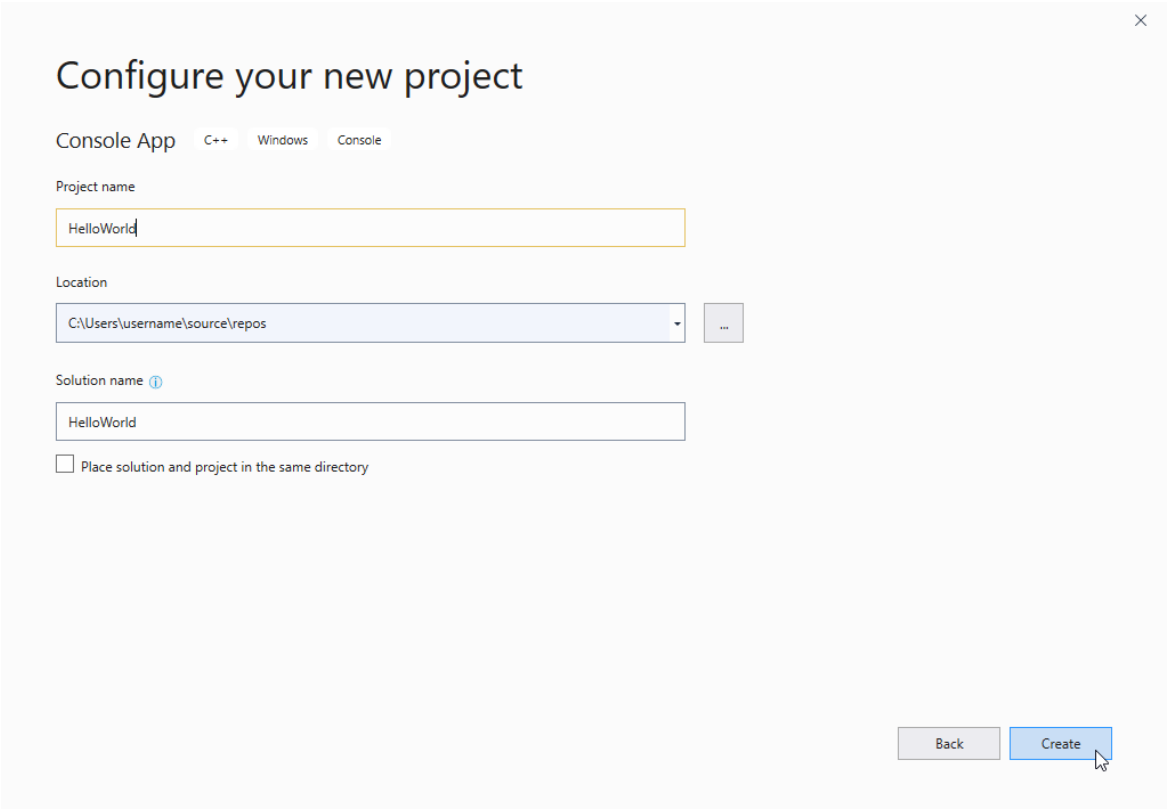
Create your app project

Visual Studio uses *projects* to organize the code for an app, and *solutions* to organize your projects. A project contains all the options, configurations, and rules used to build your apps. It manages the relationship between all the project's files and any external files. To create your app, first, you'll create a new project and solution.

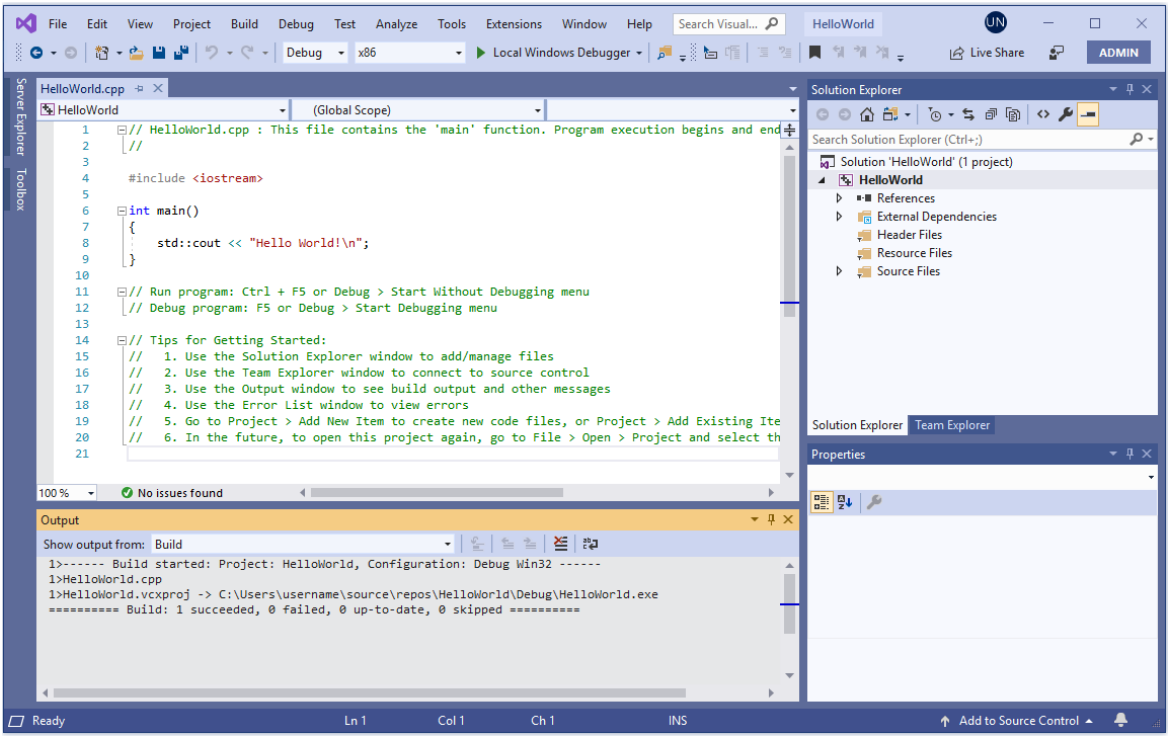
1. In Visual Studio, open the **File** menu and choose **New > Project** to open the **Create a new Project** dialog. Select the **Console App** template that has **C++**, **Windows**, and **Console** tags, and then choose **Next**.



2. In the **Configure your new project** dialog, enter *HelloWorld* in the **Project name** edit box. Choose **Create** to create the project.



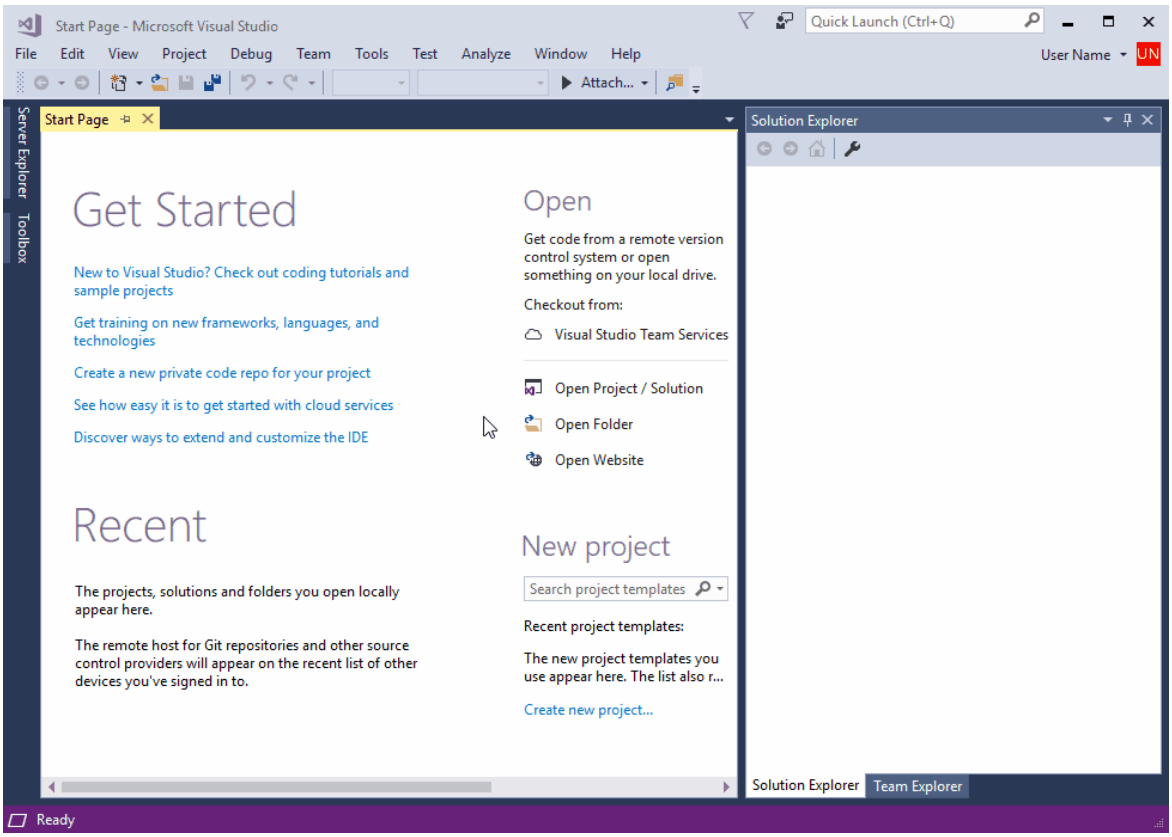
Visual Studio creates a new project. It's ready for you to add and edit your source code. By default, the Console App template fills in your source code with a "Hello World" app:



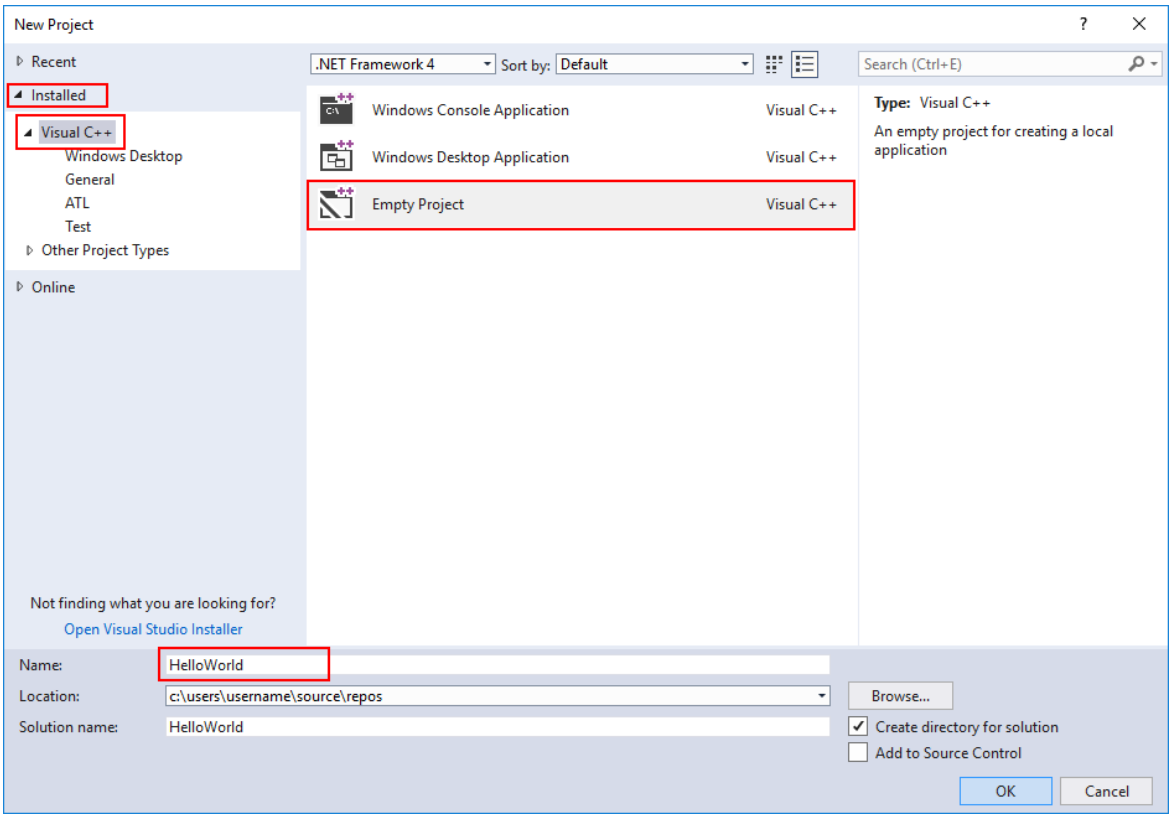
When the code looks like this in the editor, you're ready to go on to the next step and build your app.

I ran into a problem.

1. In Visual Studio, open the **File** menu and choose **New** > **Project** to open the **New Project** dialog.



2. In the **New Project** dialog, select **Installed** > **Visual C++** if it isn't selected already, and then choose the **Empty Project** template. In the **Name** field, enter *HelloWorld*. Choose **OK** to create the project.



Visual Studio creates a new, empty project. It's ready for you to specialize for the kind of app you want to create and to add your source code files. You'll do that next.

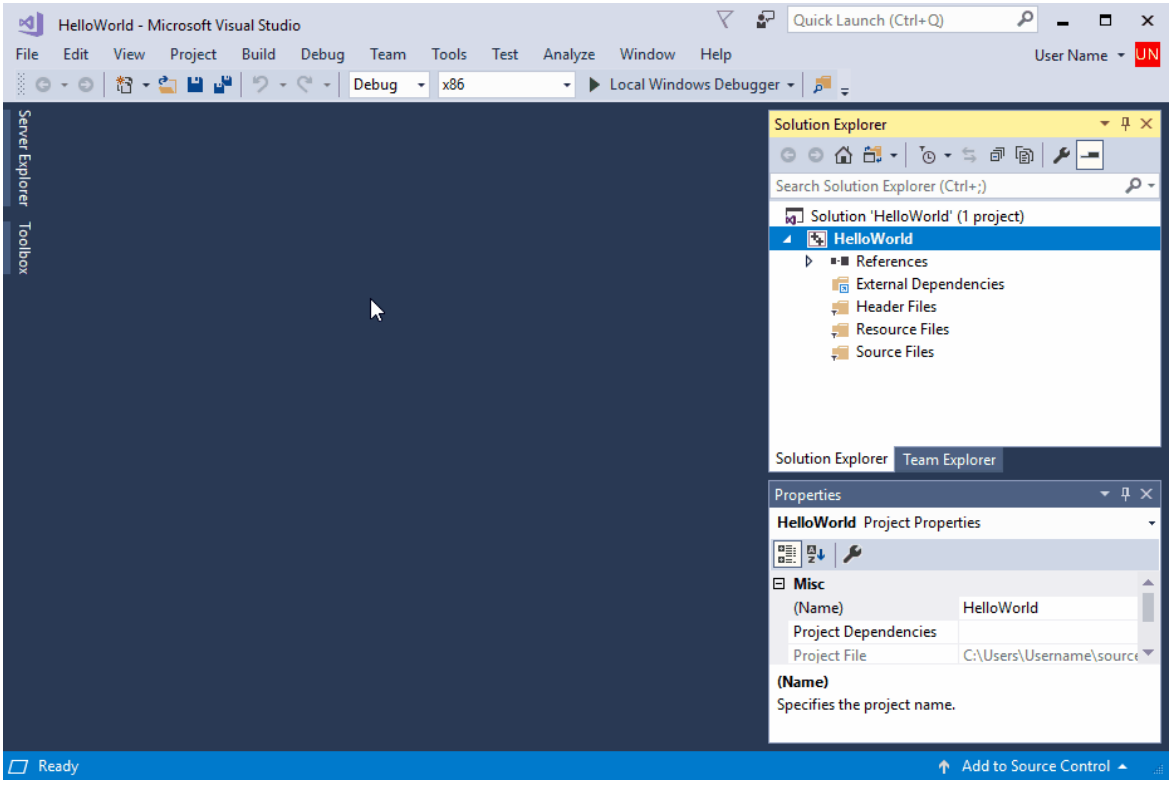
I ran into a problem.

Make your project a console app

Visual Studio can create all kinds of apps and components for Windows and other platforms. The **Empty Project** template isn't specific about what kind of app it creates. A *console app* is one that runs in a console or command prompt window. To create one, you must tell Visual Studio to build your app to use the console subsystem.

1. In Visual Studio, open the **Project** menu and choose **Properties** to open the **HelloWorld Property Pages** dialog.

2. In the **Property Pages** dialog, select **Configuration Properties > Linker > System**, and then choose the edit box next to the **Subsystem** property. In the dropdown menu that appears, select **Console (/SUBSYSTEM:CONSOLE)**. Choose **OK** to save your changes.



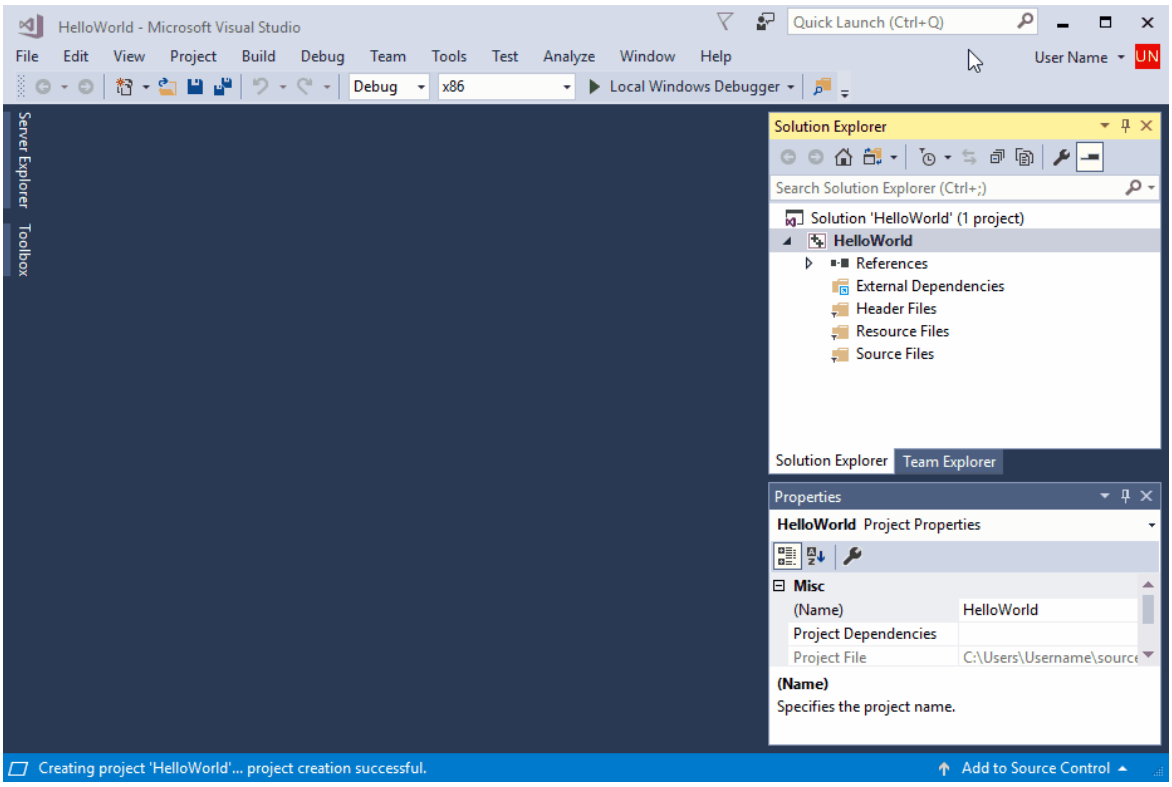
Visual Studio now knows to build your project to run in a console window. Next, you'll add a source code file and enter the code for your app.

I ran into a problem.

Add a source code file

1. In **Solution Explorer**, select the HelloWorld project. On the menu bar, choose **Project**, **Add New Item** to open the **Add New Item** dialog.

2. In the **Add New Item** dialog, select **Visual C++** under **Installed** if it isn't selected already. In the center pane, select **C++ file (.cpp)**. Change the **Name** to *HelloWorld.cpp*. Choose **Add** to close the dialog and create the file.



Visual studio creates a new, empty source code file and opens it in an editor window, ready to enter your source code.

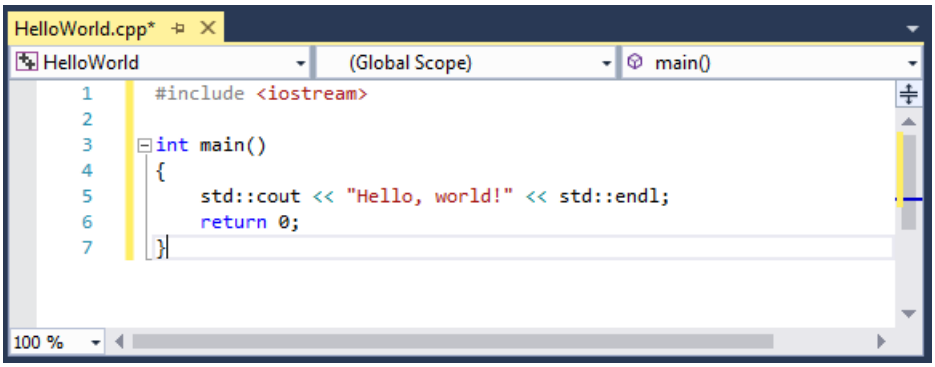
I ran into a problem.

Add code to the source file

1. Copy this code into the HelloWorld.cpp editor window.

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

The code should look like this in the editor window:



When the code looks like this in the editor, you're ready to go on to the next step and build your app.

I ran into a problem.

Next Steps

Troubleshooting guide

Come here for solutions to common issues when you create your first C++ project.

Create your app project: issues

The **New Project** dialog should show a **Console App** template that has **C++**, **Windows**, and **Console** tags. If you don't see it, there are two possible causes. It might be filtered out of the list, or it might not be installed. First, check the filter dropdowns at the top of the list of templates. Set them to **C++**, **Windows**, and **Console**. The C++ **Console App** template should appear; otherwise, the **Desktop development with C++** workload isn't installed.

To install **Desktop development with C++**, you can run the installer right from the **New Project** dialog. Choose the **Install more tools and features** link at the bottom of the template list to start the installer. If the **User Account Control** dialog requests permissions, choose **Yes**. In the installer, make sure the **Desktop development with C++** workload is checked. Then choose **Modify** to update your Visual Studio installation.

If another project with the same name already exists, choose another name for your project. Or, delete the existing project and try again. To delete an existing project, delete the solution folder (the folder that contains the *helloworld.sln* file) in File Explorer.

Go back.

If the **New Project** dialog doesn't show a **Visual C++** entry under **Installed**, your copy of Visual Studio probably doesn't have the **Desktop development with C++** workload installed. You can run the installer right from the **New Project** dialog. Choose the **Open Visual Studio Installer** link to start the installer again. If the **User Account Control** dialog requests permissions, choose **Yes**. Update the installer if necessary. In the installer, make sure the **Desktop development with C++** workload is checked, and choose **OK** to update your Visual Studio installation.

If another project with the same name already exists, choose another name for your project. Or, delete the existing project and try again. To delete an existing project, delete the solution folder (the folder that contains the *helloworld.sln* file) in File Explorer.

Go back.

Make your project a console app: issues

If you don't see **Linker** listed under **Configuration Properties**, choose **Cancel** to close the **Property Pages** dialog. Make sure that the **HelloWorld** project is selected in **Solution Explorer** before you try again. Don't select the **HelloWorld** solution, or another item, in **Solution Explorer**.

The dropdown control doesn't appear in the **SubSystem** property edit box until you select the property. Click in the edit box to select it. Or, you can press **Tab** to cycle through the dialog controls until **SubSystem** is highlighted. Choose the dropdown control or press **Alt+Down** to open it.

Go back

Add a source code file: issues

It's okay if you give the source code file a different name. However, don't add more than one file that contains the same code to your project.

If you added the wrong file type to your project, such as a header file, delete it and try again. To delete the file, select it in **Solution Explorer**. Then press the **Delete** key.

Go back.

Add code to the source file: issues

If you accidentally closed the source code file editor window, you can easily open it again. To open it, double-click on HelloWorld.cpp in the **Solution Explorer** window.

If red squiggles appear under anything in the source code editor, check that your code matches the example in spelling, punctuation, and case. Case is significant in C++ code.

Go back.

Build and run a C++ console app project

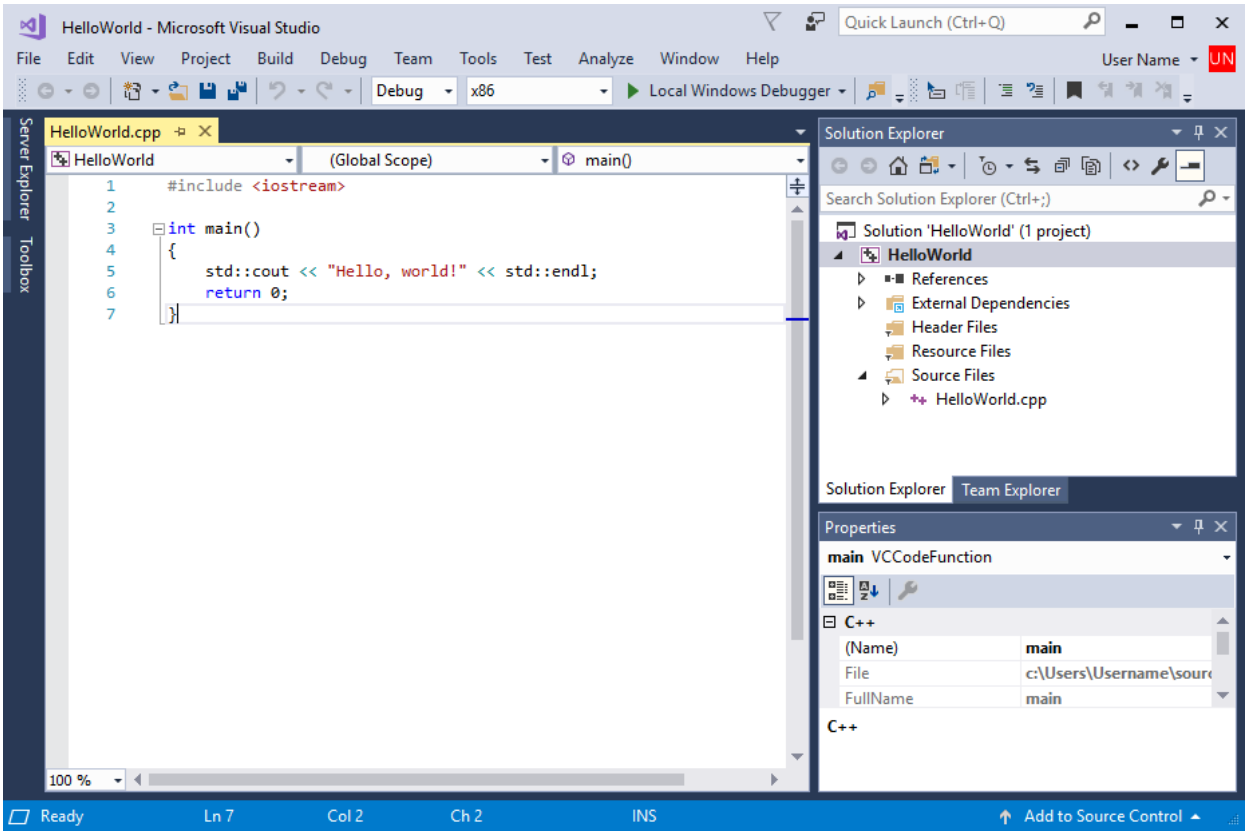
9/2/2022 • 3 minutes to read • Edit Online

You've created a C++ console app project and entered your code. Now you can build and run it within Visual Studio. Then, run it as a stand-alone app from the command line.

Prerequisites

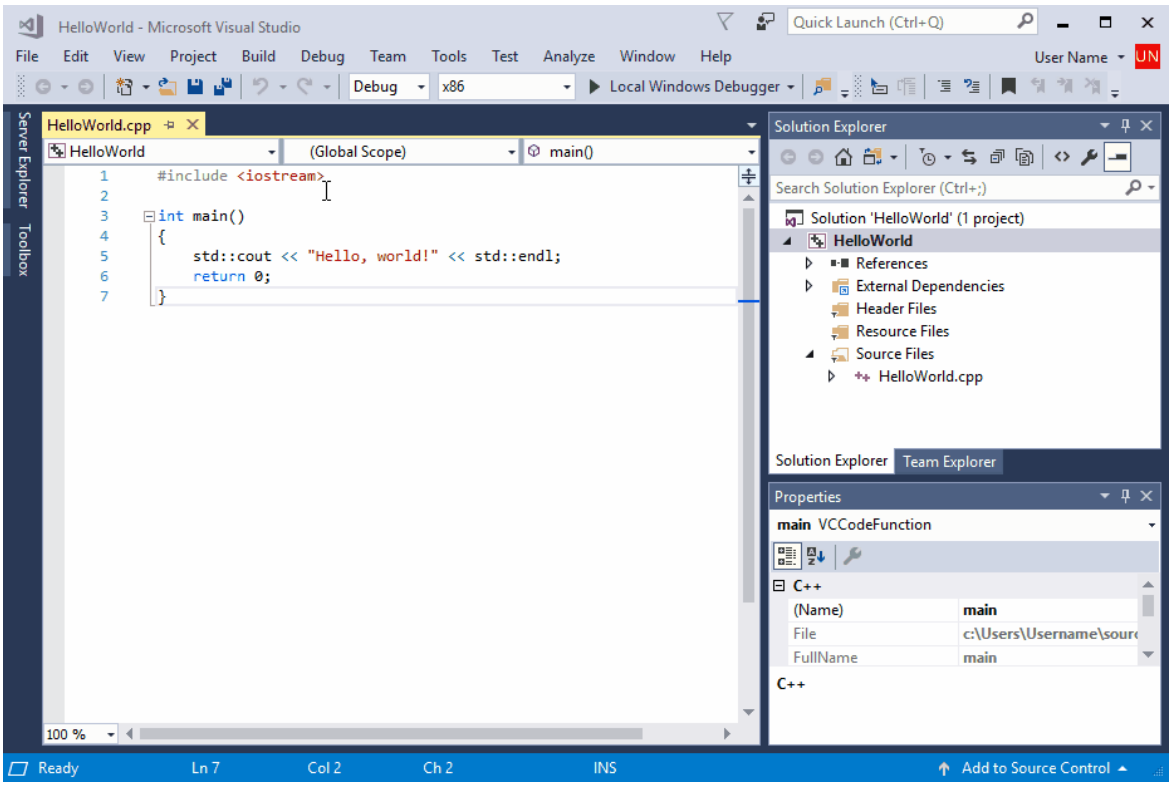
- Have Visual Studio with the Desktop development with C++ workload installed and running on your computer. If it's not installed yet, follow the steps in Install C++ support in Visual Studio.
- Create a "Hello, World!" project and enter its source code. If you haven't done this step yet, follow the steps in Create a C++ console app project.

If Visual Studio looks like this, you're ready to build and run your app:

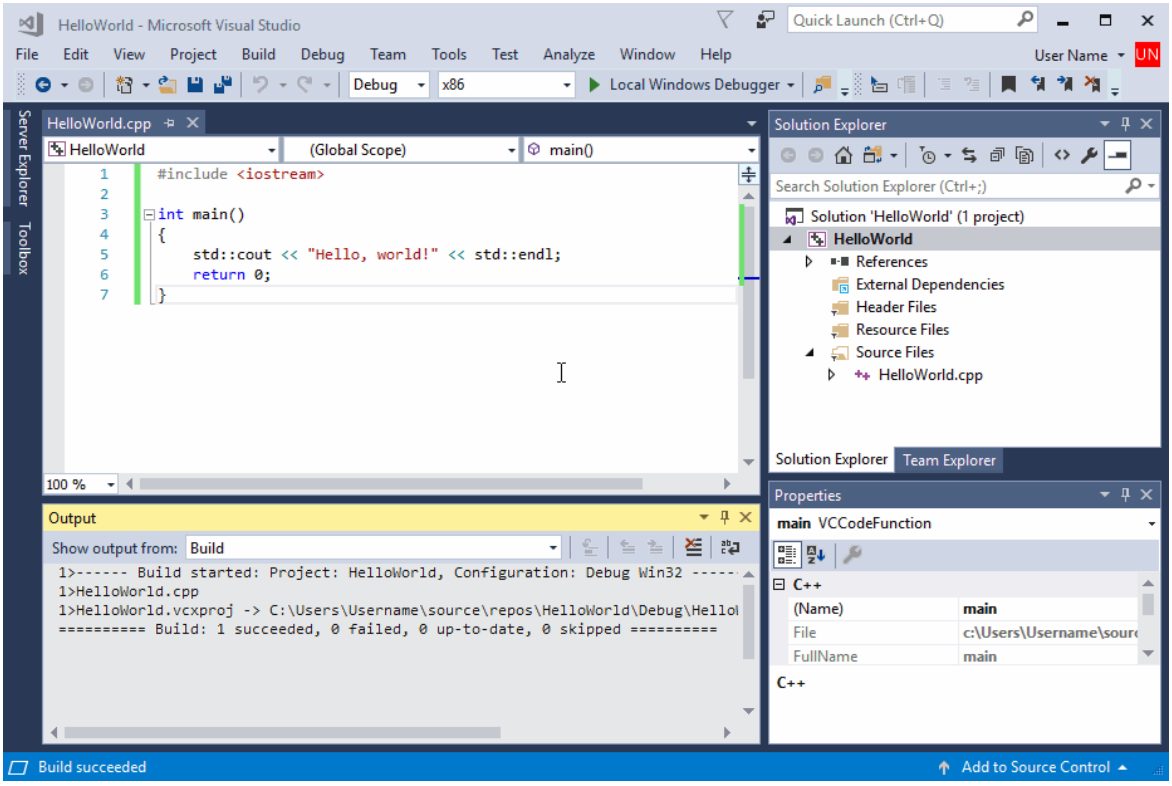


Build and run your code in Visual Studio

1. To build your project, choose **Build Solution** from the **Build** menu. The **Output** window shows the results of the build process.



2. To run the code, on the menu bar, choose **Debug**, **Start without debugging**.



A console window opens and then runs your app. When you start a console app in Visual Studio, it runs your code, then prints "Press any key to continue . . ." to give you a chance to see the output.

Congratulations! You've created your first "Hello, world!" console app in Visual Studio! Press a key to dismiss the console window and return to Visual Studio.

I ran into a problem.

Run your code in a command window

Normally, you run console apps at the command prompt, not in Visual Studio. Once your app is built by Visual

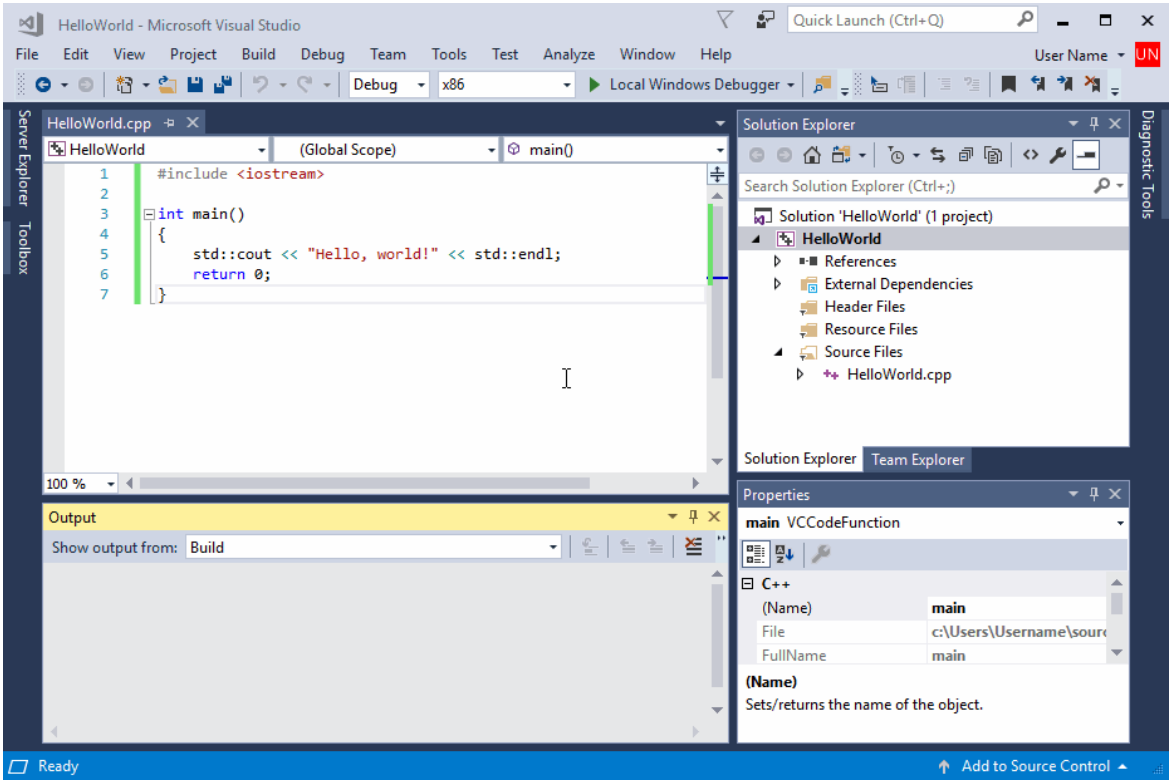
Studio, you can run it from any command window. Here's how to find and run your new app in a command prompt window.

1. In **Solution Explorer**, select the HelloWorld solution (not the HelloWorld project) and right-click to open the context menu. Choose **Open Folder in File Explorer** to open a **File Explorer** window in the HelloWorld solution folder.

2. In the **File Explorer** window, open the Debug folder. This folder contains your app, *HelloWorld.exe*, and a couple of other debugging files. Hold down the **Shift** key and right-click on *HelloWorld.exe* to open the context menu. Choose **Copy as path** to copy the path to your app to the clipboard.

3. To open a command prompt window, press **Windows+R** to open the **Run** dialog. Enter *cmd.exe* in the **Open** textbox, then choose **OK** to run a command prompt window.

4. In the command prompt window, right-click to paste the path to your app into the command prompt. Press Enter to run your app.



Congratulations, you've built and run a console app in Visual Studio!

I ran into a problem.

Next Steps

Once you've built and run this simple app, you're ready for more complex projects. For more information, see Using the Visual Studio IDE for C++ Desktop Development. It has more detailed walkthroughs that explore the capabilities of Microsoft C++ in Visual Studio.

Troubleshooting guide

Come here for solutions to common issues when you create your first C++ project.

Build and run your code in Visual Studio: issues

If red squiggles appear under anything in the source code editor, the build may have errors or warnings. Check that your code matches the example in spelling, punctuation, and case.

Go back.

Run your code in a command window: issues

If the path shown in File Explorer ends in *\HelloWorld\HelloWorld*, you've opened the HelloWorld *project* instead of the HelloWorld *solution*. You'll be confused by a Debug folder that doesn't contain your app. Navigate up a level in File Explorer to get to the solution folder, the first *HelloWorld* in the path. This folder also contains a Debug folder, and you'll find your app there.

You can also navigate to the solution Debug folder at the command line to run your app. Your app won't run from other directories without specifying the path to the app. However, you can copy your app to another directory and run it from there. It's also possible to copy it to a directory specified by your PATH environment variable, then run it from anywhere.

If you don't see **Copy as path** in the shortcut menu, dismiss the menu, and then hold down the **Shift** key while you open it again. This command is just for convenience. You can also copy the path to the folder from the File Explorer search bar, and paste it into the **Run** dialog, and then enter the name of your executable at the end. It's just a little more typing, but it has the same result.

Go back.

Welcome back to C++ - Modern C++

9/2/2022 • 8 minutes to read • Edit Online

Since its creation, C++ has become one of the most widely used programming languages in the world. Well-written C++ programs are fast and efficient. The language is more flexible than other languages: It can work at the highest levels of abstraction, and down at the level of the silicon. C++ supplies highly optimized standard libraries. It enables access to low-level hardware features, to maximize speed and minimize memory requirements. C++ can create almost any kind of program: Games, device drivers, HPC, cloud, desktop, embedded, and mobile apps, and much more. Even libraries and compilers for other programming languages get written in C++.

One of the original requirements for C++ was backward compatibility with the C language. As a result, C++ has always permitted C-style programming, with raw pointers, arrays, null-terminated character strings, and other features. They may enable great performance, but can also spawn bugs and complexity. The evolution of C++ has emphasized features that greatly reduce the need to use C-style idioms. The old C-programming facilities are still there when you need them. However, in modern C++ code you should need them less and less. Modern C++ code is simpler, safer, more elegant, and still as fast as ever.

The following sections provide an overview of the main features of modern C++. Unless noted otherwise, the features listed here are available in C++11 and later. In the Microsoft C++ compiler, you can set the `/std` compiler option to specify which version of the standard to use for your project.

Resources and smart pointers

One of the major classes of bugs in C-style programming is the *memory leak*. Leaks are often caused by a failure to call `delete` for memory that was allocated with `new`. Modern C++ emphasizes the principle of *resource acquisition is initialization* (RAII). The idea is simple. Resources (heap memory, file handles, sockets, and so on) should be *owned* by an object. That object creates, or receives, the newly allocated resource in its constructor, and deletes it in its destructor. The principle of RAII guarantees that all resources get properly returned to the operating system when the owning object goes out of scope.

To support easy adoption of RAII principles, the C++ Standard Library provides three smart pointer types: `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`. A smart pointer handles the allocation and deletion of the memory it owns. The following example shows a class with an array member that is allocated on the heap in the call to `make_unique()`. The calls to `new` and `delete` are encapsulated by the `unique_ptr` class. When a `widget` object goes out of scope, the unique_ptr destructor will be invoked and it will release the memory that was allocated for the array.

```
#include <memory>
class widget
{
private:
    std::unique_ptr<int[]> data;
public:
    widget(const int size) { data = std::make_unique<int[]>(size); }
    void do_something() {}
};

void functionUsingWidget() {
    widget w(1000000);  // lifetime automatically tied to enclosing scope
                        // constructs w, including the w.data gadget member
    // ...
    w.do_something();
    // ...
} // automatic destruction and deallocation for w and w.data
```

Whenever possible, use a smart pointer to manage heap memory. If you must use the `new` and `delete` operators explicitly, follow the principle of RAII. For more information, see Object lifetime and resource management (RAII).

`std::string` and `std::string_view`

C-style strings are another major source of bugs. By using `std::string` and `std::wstring`, you can eliminate virtually all the errors associated with C-style strings. You also gain the benefit of member functions for searching, appending, prepending, and so on. Both are highly optimized for speed. When passing a string to a function that requires only read-only access, in C++17 you can use `std::string_view` for even greater performance benefit.

`std::vector` and other Standard Library containers

The standard library containers all follow the principle of RAII. They provide iterators for safe traversal of elements. And, they're highly optimized for performance and have been thoroughly tested for correctness. By using these containers, you eliminate the potential for bugs or inefficiencies that might be introduced in custom data structures. Instead of raw arrays, use `vector` as a sequential container in C++.

```
vector<string> apples;
apples.push_back("Granny Smith");
```

Use `map` (not `unordered_map`) as the default associative container. Use `set`, `multimap`, and `multiset` for degenerate and multi cases.

```
map<string, string> apple_color;
// ...
apple_color["Granny Smith"] = "Green";
```

When performance optimization is needed, consider using:

- The `array` type when embedding is important, for example, as a class member.
- Unordered associative containers such as `unordered_map`. These have lower per-element overhead and constant-time lookup, but they can be harder to use correctly and efficiently.
- Sorted `vector`. For more information, see Algorithms.

Don't use C-style arrays. For older APIs that need direct access to the data, use accessor methods such as `f(vec.data(), vec.size());` instead. For more information about containers, see C++ Standard Library Containers.

Standard Library algorithms

Before you assume that you need to write a custom algorithm for your program, first review the C++ Standard Library algorithms. The Standard Library contains an ever-growing assortment of algorithms for many common operations such as searching, sorting, filtering, and randomizing. The math library is extensive. In C++17 and later, parallel versions of many algorithms are provided.

Here are some important examples:

- `for_each`, the default traversal algorithm (along with range-based `for` loops).
- `transform`, for not-in-place modification of container elements
- `find_if`, the default search algorithm.
- `sort`, `lower_bound`, and the other default sorting and searching algorithms.

To write a comparator, use strict `<` and use *named lambdas* when you can.

```
auto comp = [](const widget& w1, const widget& w2)
     { return w1.weight() < w2.weight(); }

sort( v.begin(), v.end(), comp );

auto i = lower_bound( v.begin(), v.end(), widget{0}, comp );
```

`auto` instead of explicit type names

C++11 introduced the `auto` keyword for use in variable, function, and template declarations. `auto` tells the compiler to deduce the type of the object so that you don't have to type it explicitly. `auto` is especially useful when the deduced type is a nested template:

```
map<int,list<string>>::iterator i = m.begin(); // C-style
auto i = m.begin(); // modern C++
```

Range-based `for` loops

C-style iteration over arrays and containers is prone to indexing errors and is also tedious to type. To eliminate these errors, and make your code more readable, use range-based `for` loops with both Standard Library containers and raw arrays. For more information, see Range-based `for` statement.

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v {1,2,3};

    // C-style
    for(int i = 0; i < v.size(); ++i)
    {
        std::cout << v[i];
    }

    // Modern C++:
    for(auto& num : v)
    {
        std::cout << num;
    }
}
```

`constexpr` expressions instead of macros

Macros in C and C++ are tokens that are processed by the preprocessor before compilation. Each instance of a macro token is replaced with its defined value or expression before the file is compiled. Macros are commonly used in C-style programming to define compile-time constant values. However, macros are error-prone and difficult to debug. In modern C++, you should prefer `constexpr` variables for compile-time constants:

```
#define SIZE 10 // C-style
constexpr int size = 10; // modern C++
```

Uniform initialization

In modern C++, you can use brace initialization for any type. This form of initialization is especially convenient when initializing arrays, vectors, or other containers. In the following example, `v2` is initialized with three instances of `S`. `v3` is initialized with three instances of `S` that are themselves initialized using braces. The compiler infers the type of each element based on the declared type of `v3`.

```
#include <vector>

struct S
{
    std::string name;
    float num;
    S(std::string s, float f) : name(s), num(f) {}
};

int main()
{
    // C-style initialization
    std::vector<S> v;
    S s1("Norah", 2.7);
    S s2("Frank", 3.5);
    S s3("Jeri", 85.9);

    v.push_back(s1);
    v.push_back(s2);
    v.push_back(s3);

    // Modern C++:
    std::vector<S> v2 {s1, s2, s3};

    // or...
    std::vector<S> v3{ {"Norah", 2.7}, {"Frank", 3.5}, {"Jeri", 85.9} };

}
```

For more information, see Brace initialization.

Move semantics

Modern C++ provides *move semantics*, which make it possible to eliminate unnecessary memory copies. In earlier versions of the language, copies were unavoidable in certain situations. A *move* operation transfers ownership of a resource from one object to the next without making a copy. Some classes own resources such as heap memory, file handles, and so on. When you implement a resource-owning class, you can define a *move constructor* and *move assignment operator* for it. The compiler chooses these special members during overload resolution in situations where a copy isn't needed. The Standard Library container types invoke the move constructor on objects if one is defined. For more information, see Move Constructors and Move Assignment Operators (C++).

Lambda expressions

In C-style programming, a function can be passed to another function by using a *function pointer*. Function pointers are inconvenient to maintain and understand. The function they refer to may be defined elsewhere in the source code, far away from the point at which it's invoked. Also, they're not type-safe. Modern C++ provides *function objects*, classes that override the `operator()` operator, which enables them to be called like a function. The most convenient way to create function objects is with inline lambda expressions. The following example shows how to use a lambda expression to pass a function object, that the `find_if` function will invoke on each element in the vector:

```
    std::vector<int> v {1,2,3,4,5};
    int x = 2;
    int y = 4;
    auto result = find_if(begin(v), end(v), [=](int i) { return i > x && i < y; });
```

The lambda expression `[=](int i) { return i > x && i < y; }` can be read as "function that takes a single

argument of type `int` and returns a boolean that indicates whether the argument is greater than `x` and less than `y`." Notice that the variables `x` and `y` from the surrounding context can be used in the lambda. The `[=]` specifies that those variables are *captured* by value; in other words, the lambda expression has its own copies of those values.

Exceptions

Modern C++ emphasizes exceptions, not error codes, as the best way to report and handle error conditions. For more information, see Modern C++ best practices for exceptions and error handling.

`std::atomic`

Use the C++ Standard Library `std::atomic` struct and related types for inter-thread communication mechanisms.

`std::variant` (C++17)

Unions are commonly used in C-style programming to conserve memory by enabling members of different types to occupy the same memory location. However, unions aren't type-safe and are prone to programming errors. C++17 introduces the `std::variant` class as a more robust and safe alternative to unions. The `std::visit` function can be used to access the members of a `variant` type in a type-safe manner.

See also

C++ Language Reference
Lambda Expressions
C++ Standard Library
Microsoft C/C++ language conformance

Create a console calculator in C++

9/2/2022 • 39 minutes to read • Edit Online

The usual starting point for a C++ programmer is a "Hello, world!" application that runs on the command line. That's what you'll create first in Visual Studio in this article, and then we'll move on to something more challenging: a calculator app.

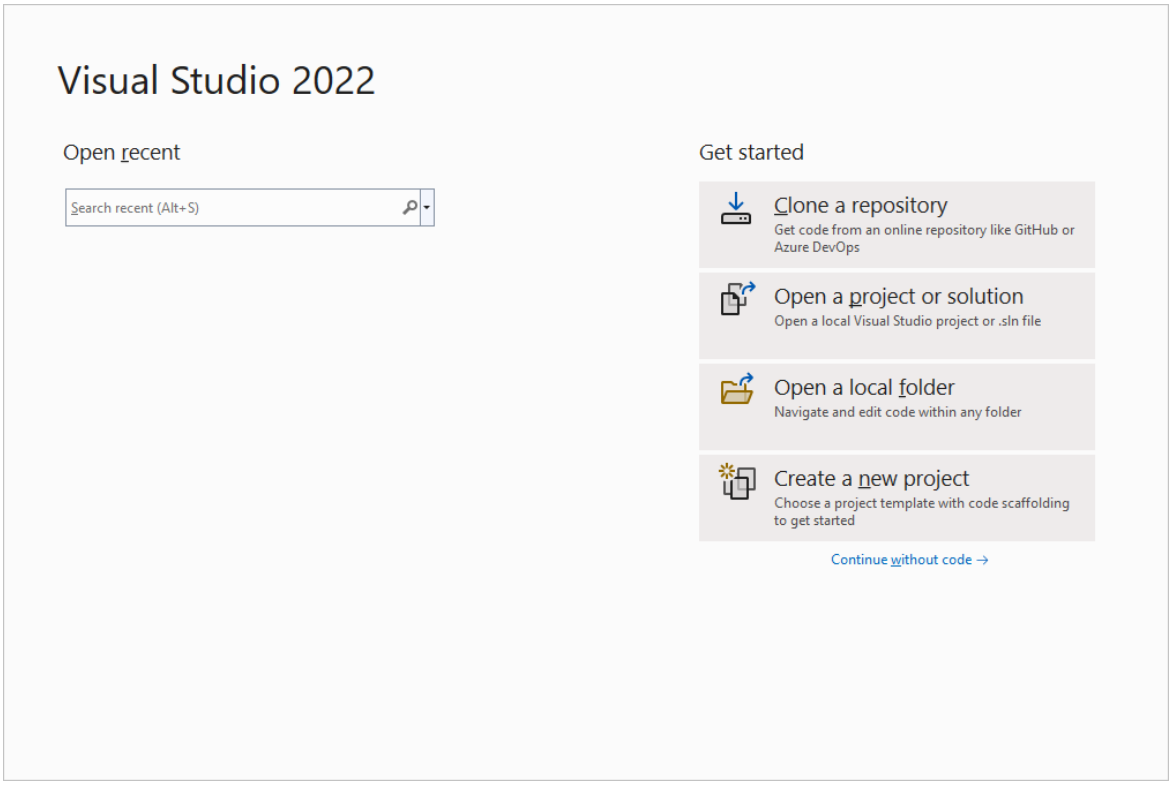
Prerequisites

- Have Visual Studio with the **Desktop development with C++** workload installed and running on your computer. If it's not installed yet, see Install C++ support in Visual Studio.

Create your app project

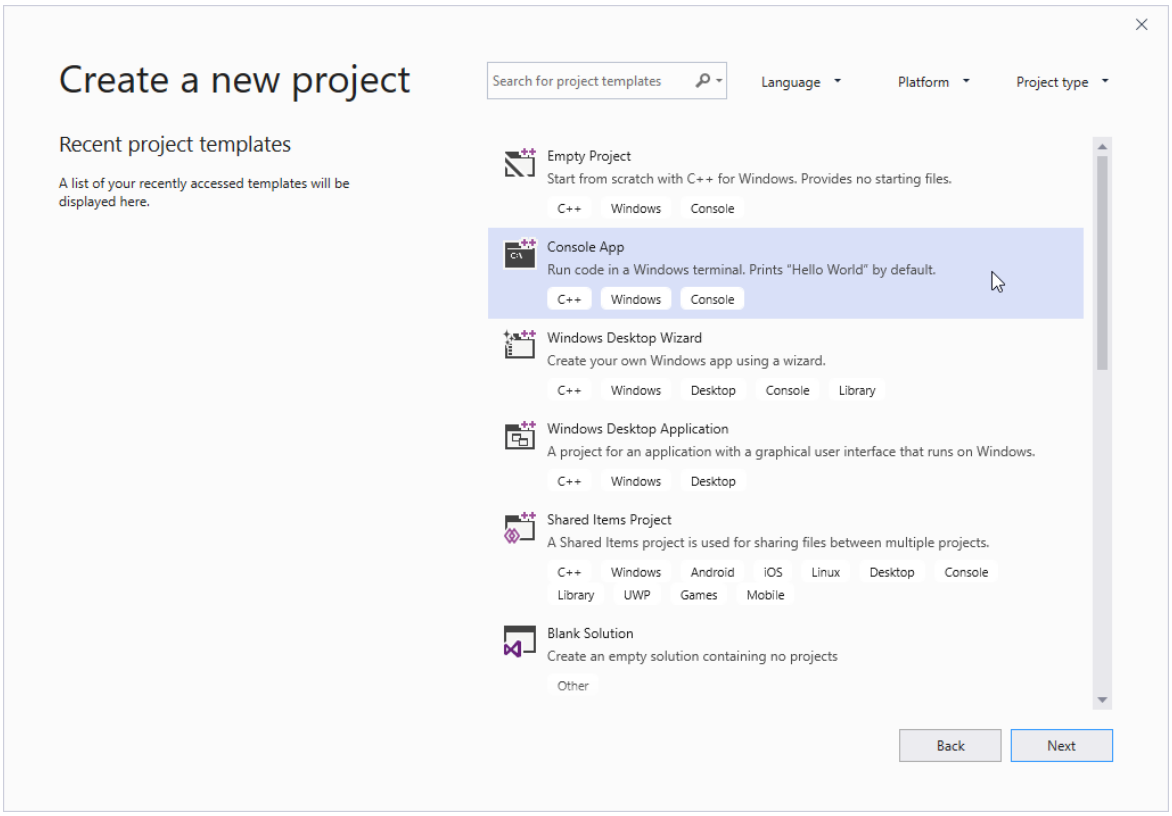
Visual Studio uses *projects* to organize the code for an app, and *solutions* to organize your projects. A project contains all the options, configurations, and rules used to build your apps. It also manages the relationship between all the project's files and any external files. To create your app, first, you'll create a new project and solution.

1. If you've just started Visual Studio, you'll see the Visual Studio Start dialog box. Choose **Create a new project** to get started.



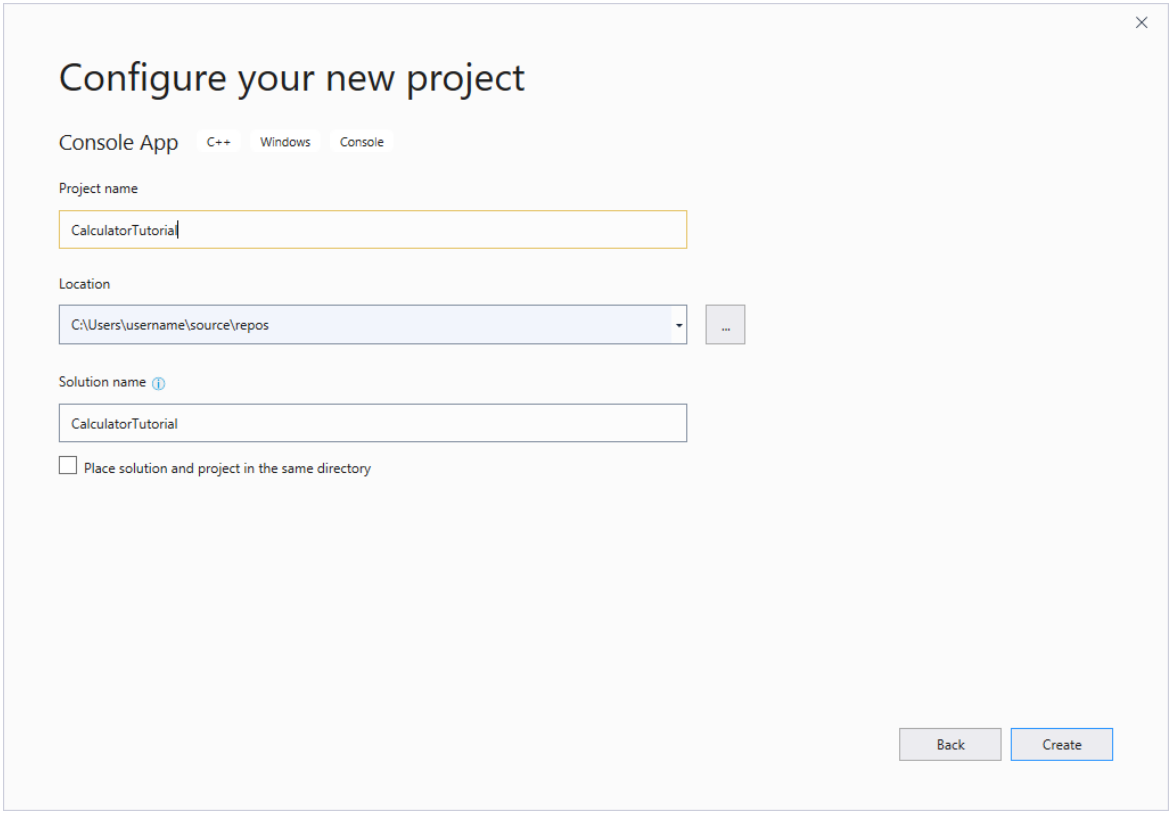
   Otherwise, on the menubar in Visual Studio, choose **File** > **New** > **Project**. The **Create a new project** window opens.

2. In the list of project templates, choose **Console App**, then choose **Next**.



> **IMPORTANT**
>
> Make sure you choose the C++ version of the **Console App** template. It has the **C++**, **Windows**, and **Console** tags, and the icon has "++" in the corner.

3. In the **Configure your new project** dialog box, select the **Project name** edit box, name your new project *CalculatorTutorial*, then choose **Create**.



An empty C++ Windows console application gets created. Console applications use a Windows console window to display output and accept user input. In Visual Studio, an editor window opens and shows the

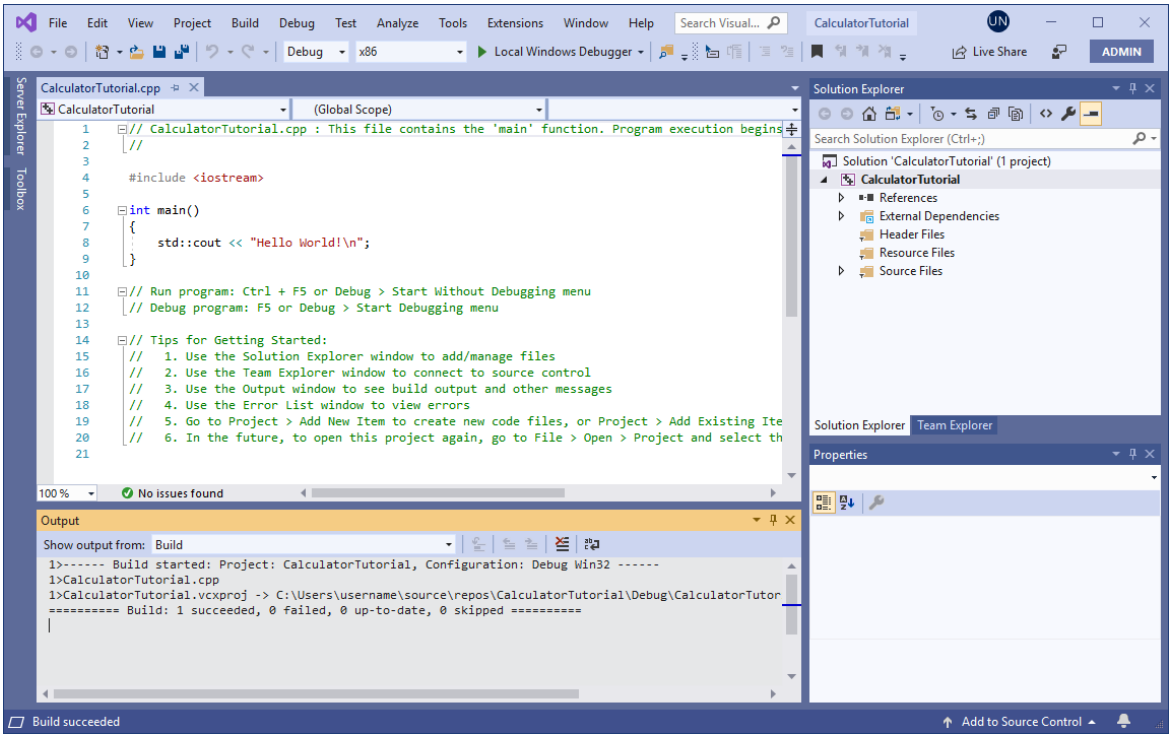
generated code:

```
// CalculatorTutorial.cpp : This file contains the 'main' function. Program execution begins and ends
there.
//

#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu

// Tips for Getting Started:
//   1. Use the Solution Explorer window to add/manage files
//   2. Use the Team Explorer window to connect to source control
//   3. Use the Output window to see build output and other messages
//   4. Use the Error List window to view errors
//   5. Go to Project > Add New Item to create new code files, or Project > Add Existing Item to add
existing code files to the project
//   6. In the future, to open this project again, go to File > Open > Project and select the .sln
file
```

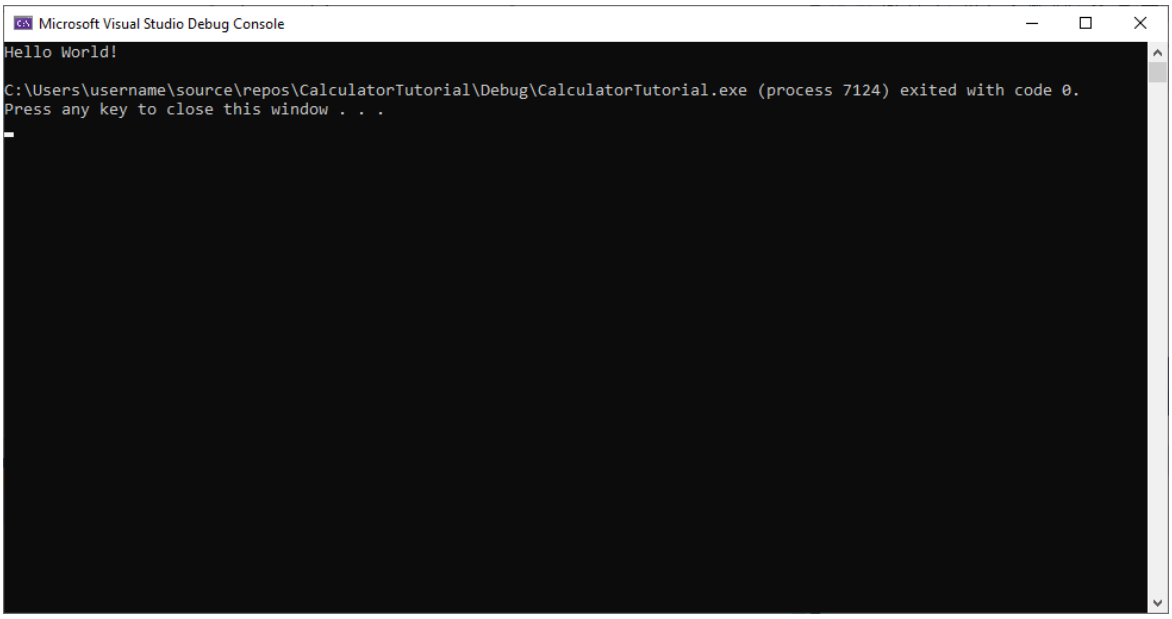
Verify that your new app builds and runs

The template for a new Windows console application creates a simple C++ "Hello World" app. At this point, you can see how Visual Studio builds and runs the apps you create right from the IDE.

1. To build your project, choose **Build Solution** from the **Build** menu. The **Output** window shows the results of the build process.



2. To run the code, on the menu bar, choose **Debug**, **Start without debugging**.



A console window opens and then runs your app. When you start a console app in Visual Studio, it runs your code, then prints "Press any key to close this window . . ." to give you a chance to see the output. Congratulations! You've created your first "Hello, world!" console app in Visual Studio!

3. Press a key to dismiss the console window and return to Visual Studio.

You now have the tools to build and run your app after every change, to verify that the code still works as you expect. Later, we'll show you how to debug it if it doesn't.

Edit the code

Now let's turn the code in this template into a calculator app.

1. In the *CalculatorTutorial.cpp* file, edit the code to match this example:

```
// CalculatorTutorial.cpp : This file contains the 'main' function. Program execution begins and ends
there.
//

#include <iostream>

using namespace std;

int main()
{
    cout << "Calculator Console Application" << endl << endl;
    cout << "Please enter the operation to perform. Format: a+b | a-b | a*b | a/b"
        << endl;
    return 0;
}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu
// Tips for Getting Started:
//   1. Use the Solution Explorer window to add/manage files
//   2. Use the Team Explorer window to connect to source control
//   3. Use the Output window to see build output and other messages
//   4. Use the Error List window to view errors
//   5. Go to Project > Add New Item to create new code files, or Project > Add Existing Item to add
existing code files to the project
//   6. In the future, to open this project again, go to File > Open > Project and select the .sln
file
```

Understanding the code:

- The `#include` statements allow you to reference code located in other files. Sometimes, you may see a filename surrounded by angle brackets (< >); other times, it's surrounded by quotes (" "). In general, angle brackets are used when referencing the C++ Standard Library, while quotes are used for other files.
- The `using namespace std;` line tells the compiler to expect stuff from the C++ Standard Library to be used in this file. Without this line, each keyword from the library would have to be preceded with a `std::`, to denote its scope. For instance, without that line, each reference to `cout` would have to be written as `std::cout`. The `using` statement is added to make the code look more clean.
- The `cout` keyword is used to print to standard output in C++. The << operator tells the compiler to send whatever is to the right of it to the standard output.
- The **endl** keyword is like the Enter key; it ends the line and moves the cursor to the next line. It is a better practice to put a `\n` inside the string (contained by "") to do the same thing, as `endl` always flushes the buffer and can hurt the performance of the program, but since this is a very small app, `endl` is used instead for better readability.
- All C++ statements must end with semicolons and all C++ applications must contain a `main()` function. This function is what the program runs at the start. All code must be accessible from `main()` in order to be used.

2. To save the file, enter **Ctrl+S**, or choose the **Save** icon near the top of the IDE, the floppy disk icon in the toolbar under the menu bar.

3. To run the application, press **Ctrl+F5** or go to the **Debug** menu and choose **Start Without Debugging**. You should see a console window appear that displays the text specified in the code.

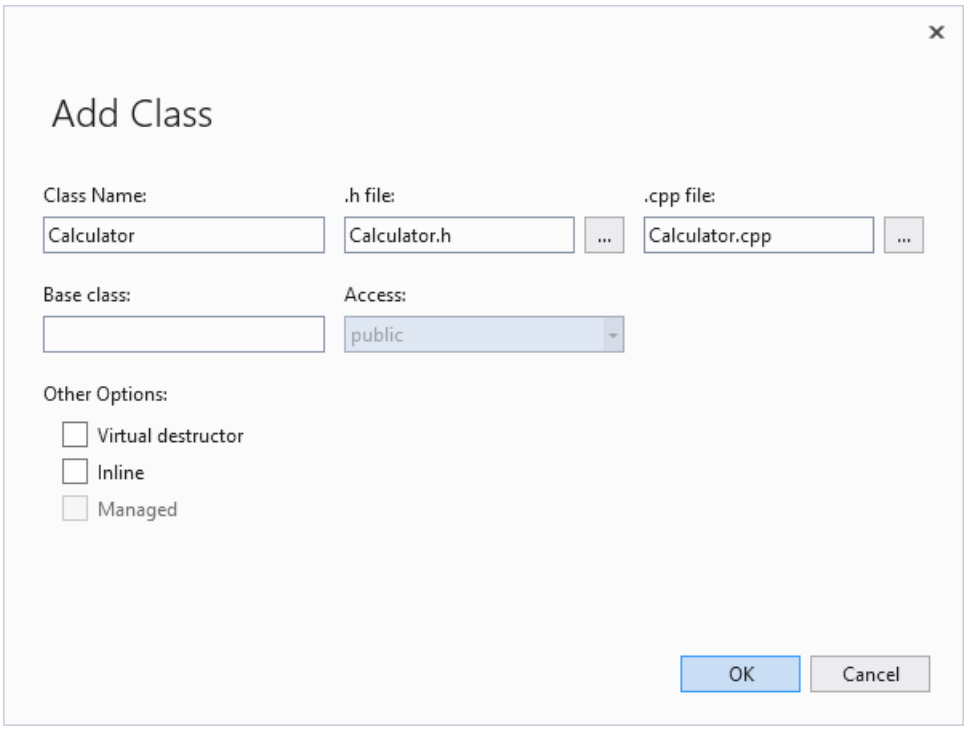
4. Close the console window when you're done.

Add code to do some math

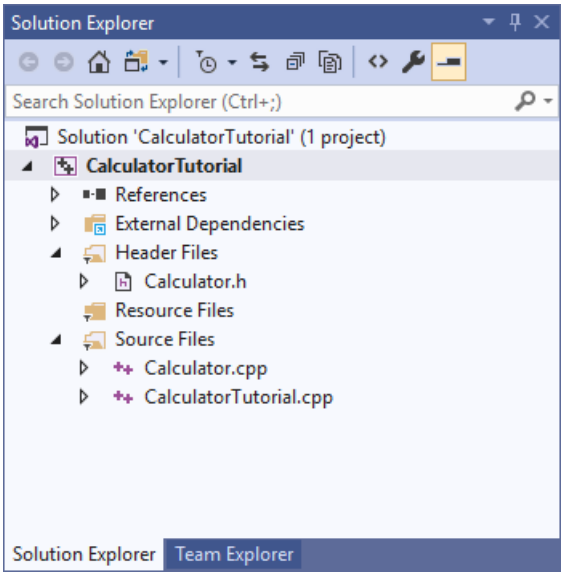
It's time to add some math logic.

To add a Calculator class

1. Go to the **Project** menu and choose **Add Class**. In the **Class Name** edit box, enter *Calculator*. Choose **OK**. Two new files get added to your project. To save all your changed files at once, press **Ctrl+Shift+S**. It's a keyboard shortcut for **File** > **Save All**. There's also a toolbar button for **Save All**, an icon of two floppy disks, found beside the **Save** button. In general, it's good practice to do **Save All** frequently, so you don't miss any files when you save.



A class is like a blueprint for an object that does something. In this case, we define a calculator and how it should work. The **Add Class** wizard you used above created .h and .cpp files that have the same name as the class. You can see a full list of your project files in the **Solution Explorer** window, visible on the side of the IDE. If the window isn't visible, you can open it from the menu bar: choose **View** > **Solution Explorer**.



You should now have three tabs open in the editor: *CalculatorTutorial.cpp*, *Calculator.h*, and *Calculator.cpp*. If you accidentally close one of them, you can reopen it by double-clicking it in the **Solution Explorer** window.

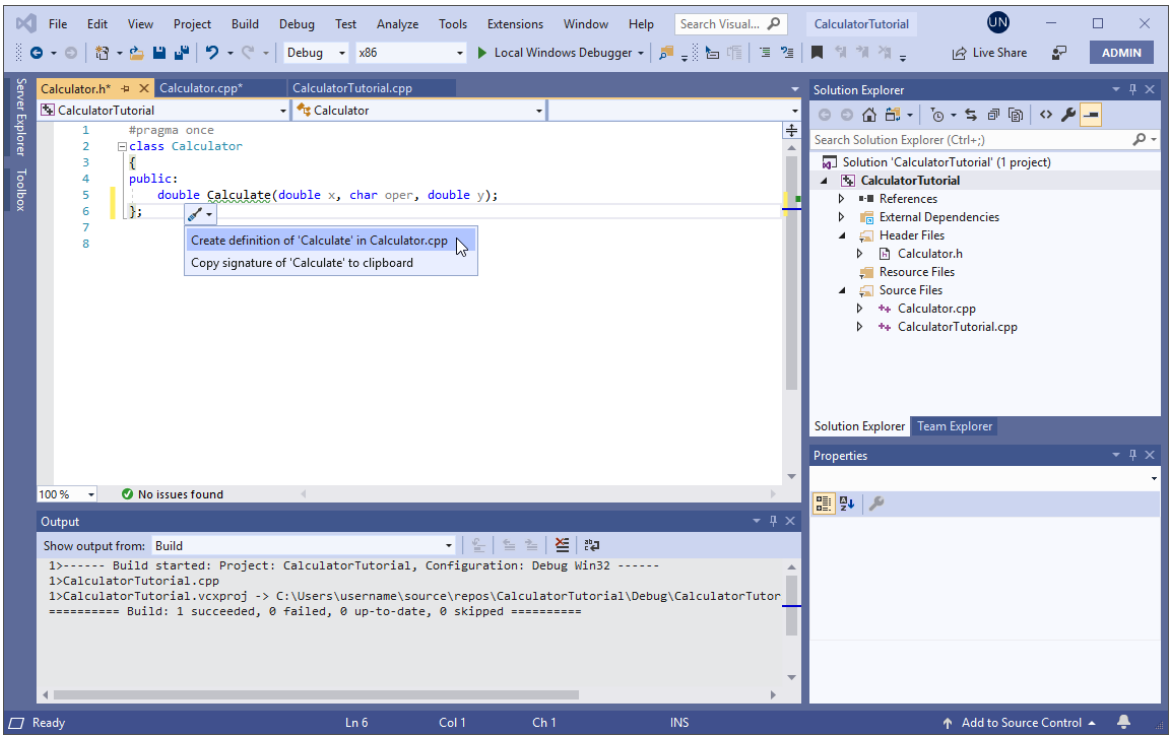
2. In **Calculator.h**, remove the `Calculator();` and `~Calculator();` lines that were generated, since you won't need them here. Next, add the following line of code so the file now looks like this:

```
#pragma once
class Calculator
{
public:
    double Calculate(double x, char oper, double y);
};
```

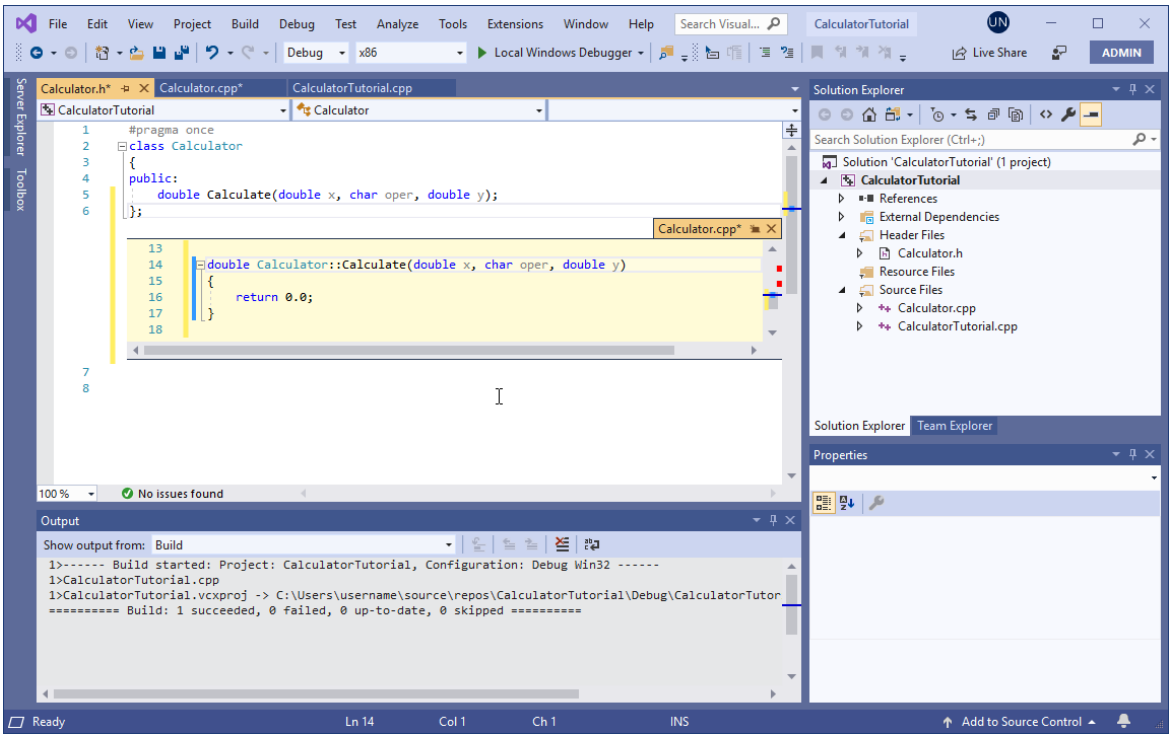
Understanding the code

- The line you added declares a new function called `Calculate`, which we'll use to run math operations for addition, subtraction, multiplication, and division.
- C++ code is organized into *header* (.h) files and *source* (.cpp) files. Several other file extensions are supported by various compilers, but these are the main ones to know about. Functions and variables are normally *declared*, that is, given a name and a type, in header files, and *implemented*, or given a definition, in source files. To access code defined in another file, you can use `#include "filename.h"`, where 'filename.h' is the name of the file that declares the variables or functions you want to use.
- The two lines you deleted declared a *constructor* and *destructor* for the class. For a simple class like this one, the compiler creates them for you, and their uses are beyond the scope of this tutorial.
- It's good practice to organize your code into different files based on what it does, so it's easy to find the code you need later. In our case, we define the `Calculator` class separately from the file containing the `main()` function, but we plan to reference the `Calculator` class in `main()`.

3. You'll see a green squiggle appear under `Calculate`. It's because we haven't defined the `Calculate` function in the .cpp file. Hover over the word, click the lightbulb (in this case, a screwdriver) that pops up, and choose **Create definition of 'Calculate' in Calculator.cpp**.



A pop-up appears that gives you a peek of the code change that was made in the other file. The code was added to *Calculator.cpp*.



Currently, it just returns 0.0. Let's change that. Press **Esc** to close the pop-up.

4. Switch to the *Calculator.cpp* file in the editor window. Remove the `Calculator()` and `~Calculator()` sections (as you did in the .h file) and add the following code to `Calculate()`:

```
#include "Calculator.h"

double Calculator::Calculate(double x, char oper, double y)
{
    switch(oper)
    {
        case '+':
            return x + y;
        case '-':
            return x - y;
        case '*':
            return x * y;
        case '/':
            return x / y;
        default:
            return 0.0;
    }
}
```

Understanding the code

- The function `Calculate` consumes a number, an operator, and a second number, then performs the requested operation on the numbers.
- The switch statement checks which operator was provided, and only executes the case corresponding to that operation. The default: case is a fallback in case the user types an operator that isn't accepted, so the program doesn't break. In general, it's best to handle invalid user input in a more elegant way, but this is beyond the scope of this tutorial.
- The `double` keyword denotes a type of number that supports decimals. This way, the calculator can handle both decimal math and integer math. The `Calculate` function is required to always return such a number due to the `double` at the very start of the code (this denotes the function's return type), which is why we return 0.0 even in the default case.
- The .h file declares the function *prototype*, which tells the compiler upfront what parameters it

requires, and what return type to expect from it. The .cpp file has all the implementation details of the function.

If you build and run the code again at this point, it will still exit after asking which operation to perform. Next, you'll modify the `main` function to do some calculations.

To call the Calculator class member functions

1. Now let's update the `main` function in *CalculatorTutorial.cpp*:

```
// CalculatorTutorial.cpp : This file contains the 'main' function. Program execution begins and ends
there.
//

#include <iostream>
#include "Calculator.h"

using namespace std;

int main()
{
    double x = 0.0;
    double y = 0.0;
    double result = 0.0;
    char oper = '+';

    cout << "Calculator Console Application" << endl << endl;
    cout << "Please enter the operation to perform. Format: a+b | a-b | a*b | a/b"
         << endl;

    Calculator c;
    while (true)
    {
        cin >> x >> oper >> y;
        result = c.Calculate(x, oper, y);
        cout << "Result is: " << result << endl;
    }

    return 0;
}
```

Understanding the code

- Since C++ programs always start at the `main()` function, we need to call our other code from there, so a `#include` statement is needed.
- Some initial variables `x`, `y`, `oper`, and `result` are declared to store the first number, second number, operator, and final result, respectively. It is always good practice to give them some initial values to avoid undefined behavior, which is what is done here.
- The `Calculator c;` line declares an object named 'c' as an instance of the `Calculator` class. The class itself is just a blueprint for how calculators work; the object is the specific calculator that does the math.
- The `while (true)` statement is a loop. The code inside the loop continues to execute over and over again as long as the condition inside the `()` holds true. Since the condition is simply listed as `true`, it's always true, so the loop runs forever. To close the program, the user must manually close the console window. Otherwise, the program always waits for new input.
- The `cin` keyword is used to accept input from the user. This input stream is smart enough to process a line of text entered in the console window and place it inside each of the variables listed, in order, assuming the user input matches the required specification. You can modify this line to accept different types of input, for instance, more than two numbers, though the `Calculate()`

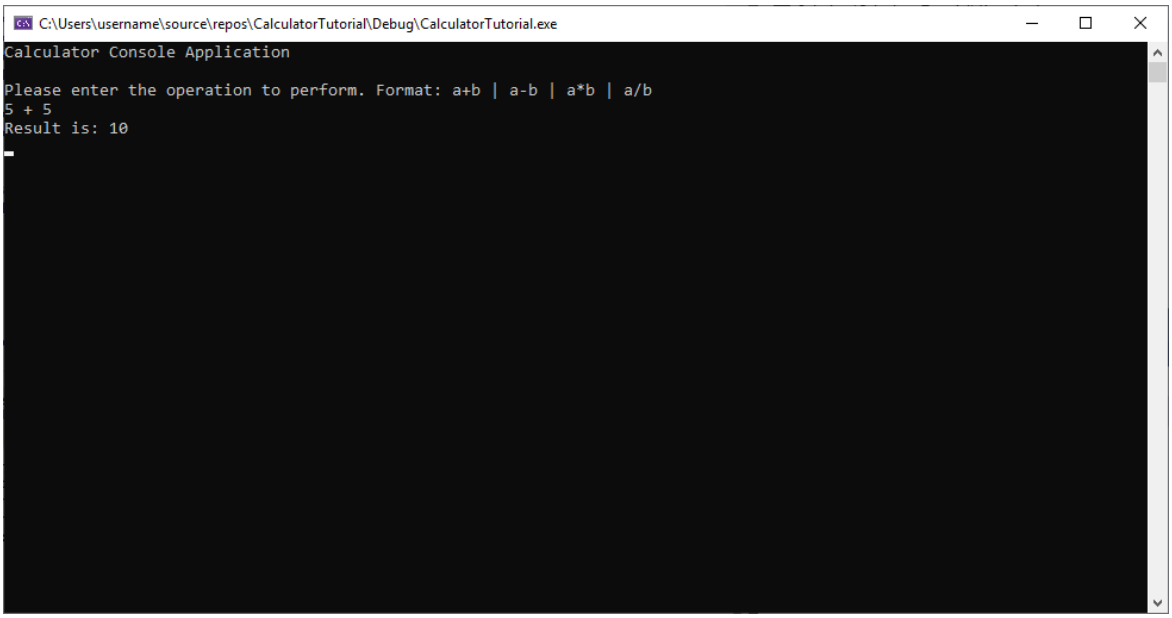
function would also need to be updated to handle this.

- The `c.Calculate(x, oper, y);` expression calls the `Calculate` function defined earlier, and supplies the entered input values. The function then returns a number that gets stored in `result`.
- Finally, `result` is printed to the console, so the user sees the result of the calculation.

Build and test the code again

Now it's time to test the program again to make sure everything works properly.

1. Press **Ctrl+F5** to rebuild and start the app.
2. Enter `5 + 5`, and press **Enter**. Verify that the result is 10.

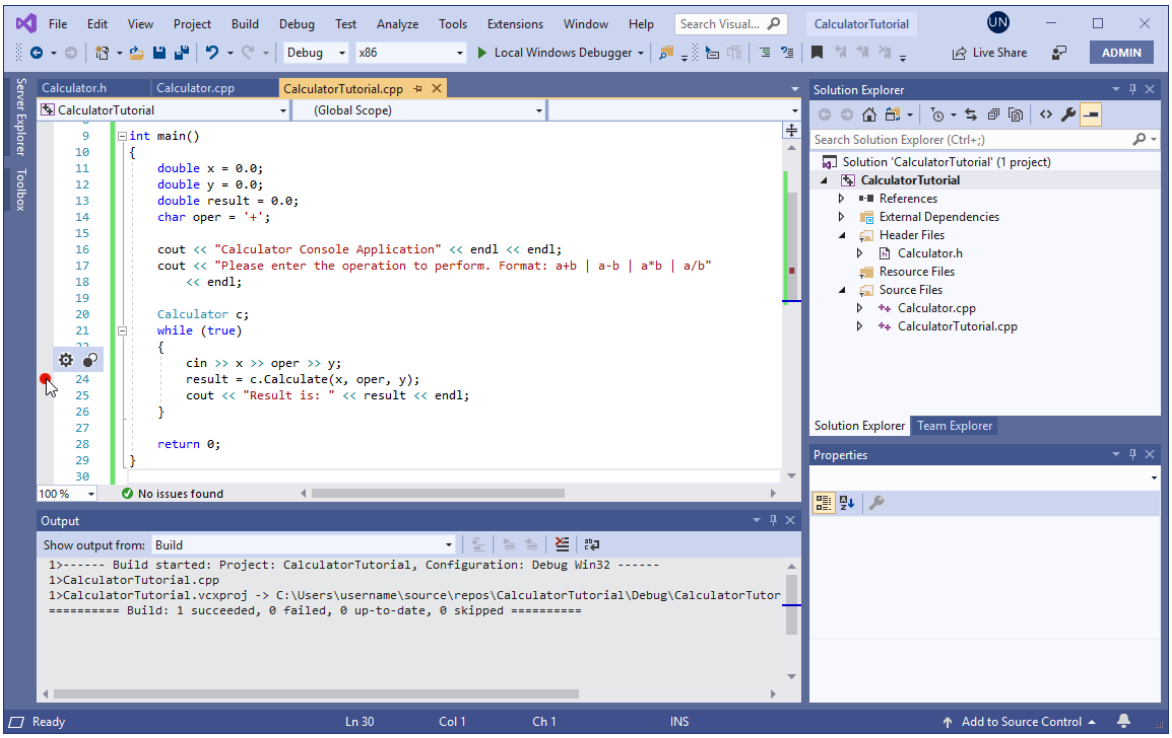


Debug the app

Since the user is free to type anything into the console window, let's make sure the calculator handles some input as expected. Instead of running the program, let's debug it instead, so we can inspect what it's doing in detail, step-by-step.

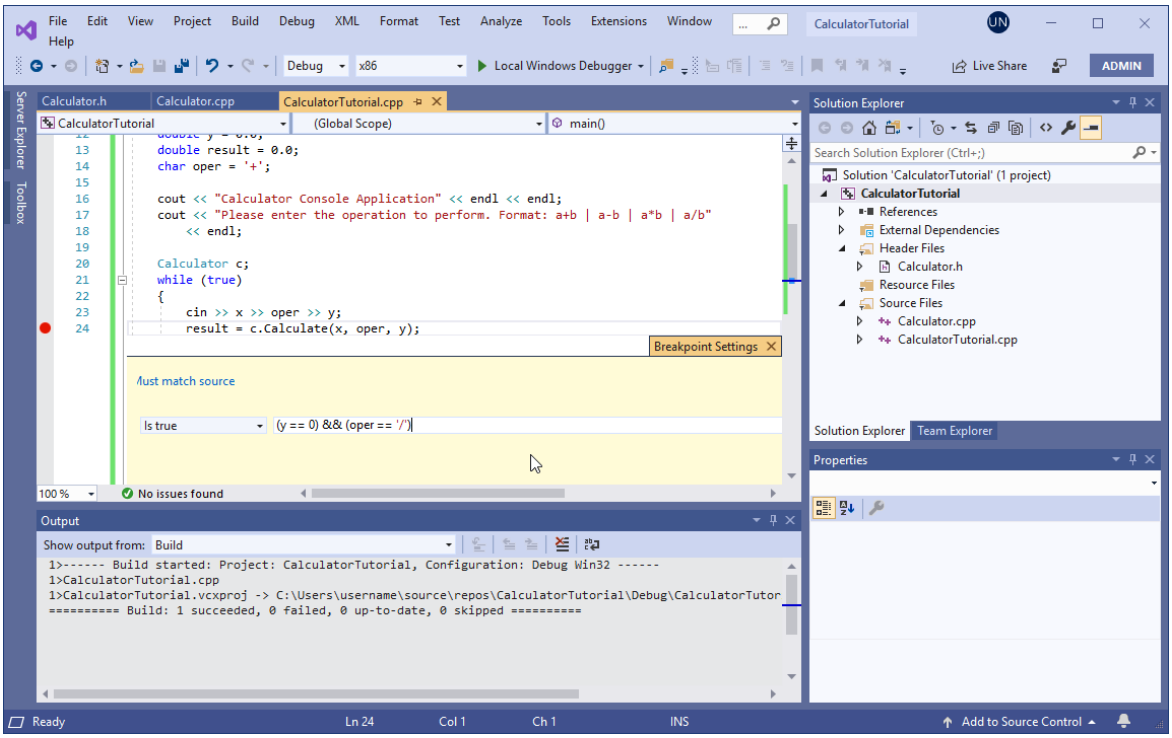
To run the app in the debugger

1. Set a breakpoint on the `result = c.Calculate(x, oper, y);` line, just after the user was asked for input. To set the breakpoint, click next to the line in the gray vertical bar along the left edge of the editor window. A red dot appears.



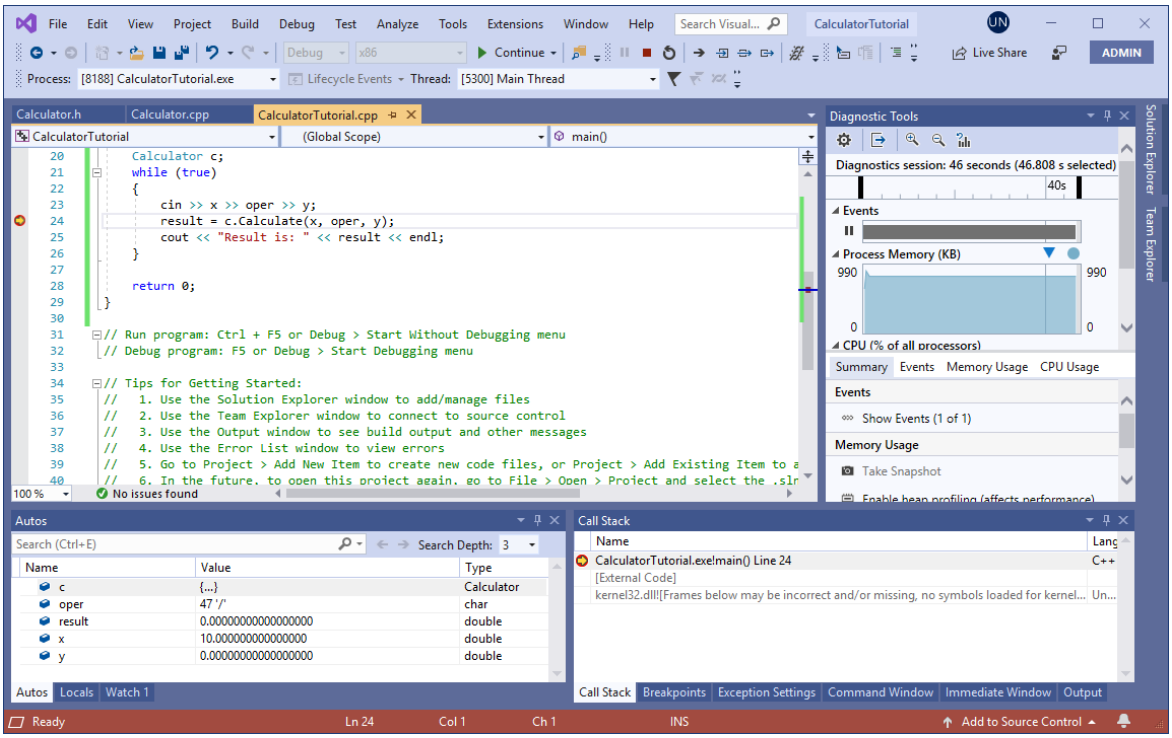
Now when we debug the program, it always pauses execution at that line. We already have a rough idea that the program works for simple cases. Since we don't want to pause execution every time, let's make the breakpoint conditional.

2. Right-click the red dot that represents the breakpoint, and choose **Conditions**. In the edit box for the condition, enter `(y == 0) && (oper == '/')`. Choose the **Close** button when you're done. The condition is saved automatically.



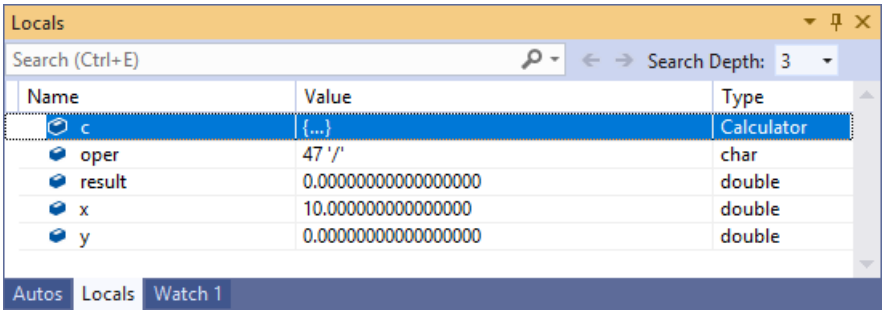
Now we pause execution at the breakpoint specifically if a division by 0 is attempted.

3. To debug the program, press **F5**, or choose the **Local Windows Debugger** toolbar button that has the green arrow icon. In your console app, if you enter something like "5 - 0", the program behaves normally and keeps running. However, if you type "10 / 0", it pauses at the breakpoint. You can even put any number of spaces between the operator and numbers: `cin` is smart enough to parse the input appropriately.

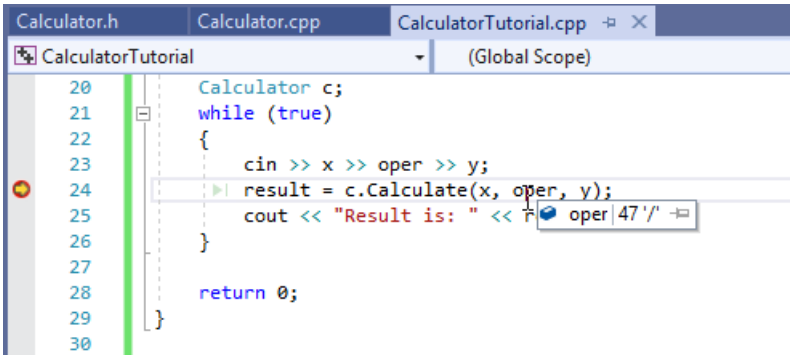


Useful windows in the debugger

Whenever you debug your code, you may notice that some new windows appear. These windows can assist your debugging experience. Take a look at the **Autos** window. The **Autos** window shows you the current values of variables used at least three lines before and up to the current line. To see all of the variables from that function, switch to the **Locals** window. You can actually modify the values of these variables while debugging, to see what effect they would have on the program. In this case, we'll leave them alone.



You can also just hover over variables in the code itself to see their current values where the execution is currently paused. Make sure the editor window is in focus by clicking on it first.



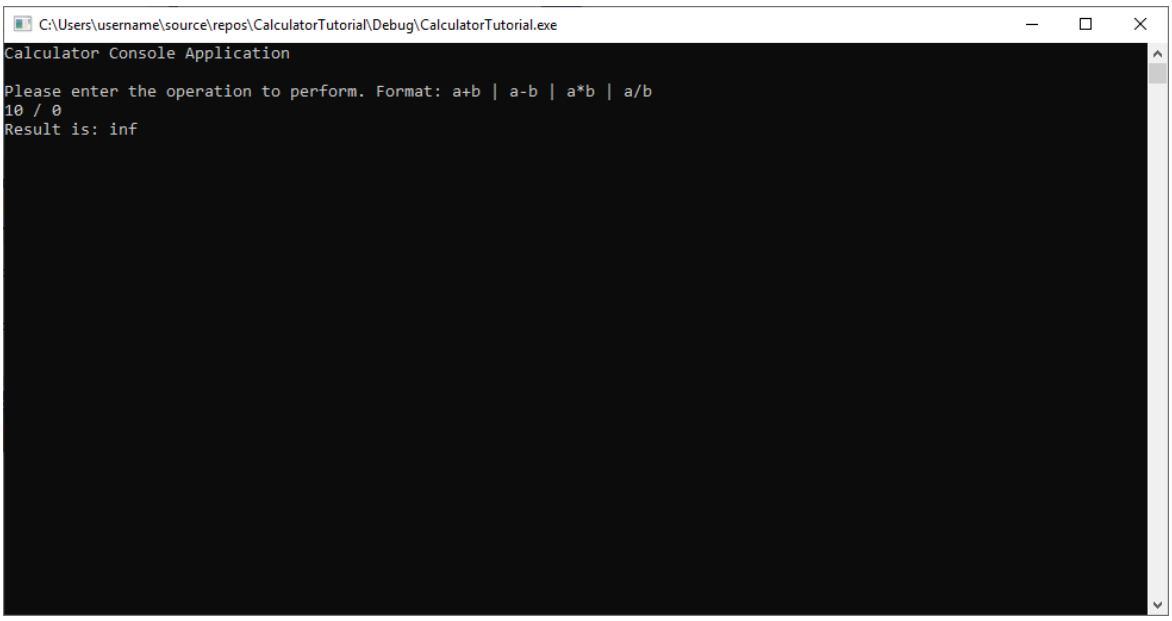
To continue debugging

1. The yellow line on the left shows the current point of execution. The current line calls `Calculate`, so press **F11** to **Step Into** the function. You'll find yourself in the body of the `Calculate` function. Be careful with **Step Into**; if you do it too much, you may waste a lot of time. It goes into any code you use on the line you are on, including standard library functions.

2. Now that the point of execution is at the start of the `Calculate` function, press **F10** to move to the next line in the program's execution. **F10** is also known as **Step Over**. You can use **Step Over** to move from line to line, without delving into the details of what is occurring in each part of the line. In general you should use **Step Over** instead of **Step Into**, unless you want to dive more deeply into code that is being called from elsewhere (as you did to reach the body of `Calculate`).

3. Continue using **F10** to **Step Over** each line until you get back to the `main()` function in the other file, and stop on the `cout` line.

   It looks like the program is doing what is expected: it takes the first number, and divides it by the second. On the `cout` line, hover over the `result` variable or take a look at `result` in the **Autos** window. You'll see its value is listed as "inf", which doesn't look right, so let's fix it. The `cout` line just outputs whatever value is stored in `result`, so when you step one more line forward using **F10**, the console window displays:



   This result happens because division by zero is undefined, so the program doesn't have a numerical answer to the requested operation.

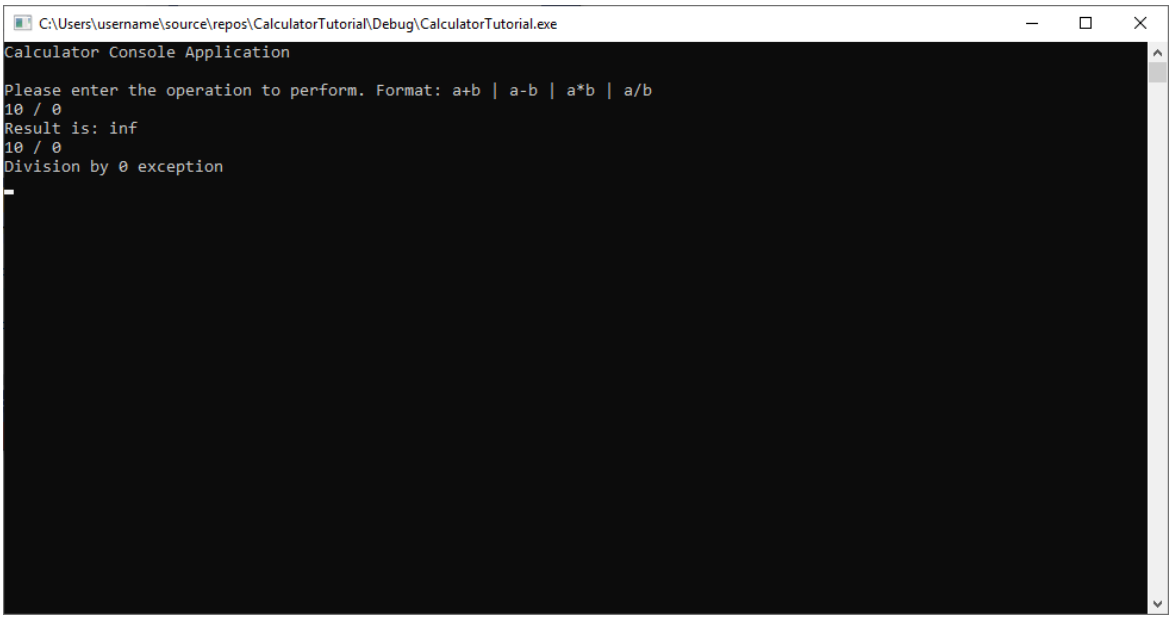
To fix the "divide by zero" error

Let's handle division by zero more gracefully, so a user can understand the problem.

1. Make the following changes in *CalculatorTutorial.cpp*. (You can leave the program running as you edit, thanks to a debugger feature called **Edit and Continue**):

```
// CalculatorTutorial.cpp : This file contains the 'main' function. Program execution begins and ends
there.
//

#include <iostream>
#include "Calculator.h"

using namespace std;

int main()
{
    double x = 0.0;
    double y = 0.0;
    double result = 0.0;
    char oper = '+';

    cout << "Calculator Console Application" << endl << endl;
    cout << "Please enter the operation to perform. Format: a+b | a-b | a*b | a/b" << endl;

    Calculator c;
    while (true)
    {
        cin  >> x  >> oper  >> y;
        if (oper == '/' && y == 0)
        {
            cout << "Division by 0 exception" << endl;
            continue;
        }
        else
        {
            result = c.Calculate(x, oper, y);
        }
        cout << "Result is: " << result << endl;
    }

    return 0;
}
```

2. Now press **F5** once. Program execution continues all the way until it has to pause to ask for user input. Enter `10 / 0` again. Now, a more helpful message is printed. The user is asked for more input, and the program continues executing normally.



NOTE

When you edit code while in debugging mode, there is a risk of code becoming stale. This happens when the debugger is still running your old code, and has not yet updated it with your changes. The debugger pops up a dialog to inform you when this happens. Sometimes, you may need to press **F5** to refresh the code being executed. In particular, if you make a change inside a function while the point of execution is inside that function, you'll need to step out of the function, then back into it again to get the updated code. If that doesn't work for some reason and you see an error message, you can stop debugging by clicking on the red square in the toolbar under the menus at the top of the IDE, then start debugging again by entering **F5** or by choosing the green "play" arrow beside the stop button on the toolbar.

Understanding the Run and Debug shortcuts

- **F5** (or **Debug** > **Start Debugging**) starts a debugging session if one isn't already active, and runs the program until a breakpoint is hit or the program needs user input. If no user input is needed and no breakpoint is available to hit, the program terminates and the console window closes itself when the program finishes running. If you have something like a "Hello World" program to run, use **Ctrl+F5** or set a breakpoint before you enter **F5** to keep the window open.
- **Ctrl+F5** (or **Debug** > **Start Without Debugging**) runs the application without going into debug mode. This is slightly faster than debugging, and the console window stays open after the program finishes executing.
- **F10** (known as **Step Over**) lets you iterate through code, line-by-line, and visualize how the code is run and what variable values are at each step of execution.
- **F11** (known as **Step Into**) works similarly to **Step Over**, except it steps into any functions called on the line of execution. For example, if the line being executed calls a function, pressing **F11** moves the pointer into the body of the function, so you can follow the function's code being run before coming back to the line you started at. Pressing **F10** steps over the function call and just moves to the next line; the function call still happens, but the program doesn't pause to show you what it's doing.

Close the app

- If it's still running, close the console window for the calculator app.

Add Git source control

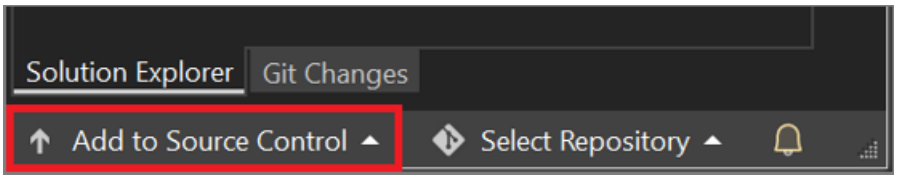
Now that you've created an app, you might want to add it to a Git repository. We've got you covered. Visual Studio makes that process easy with Git tools you can use directly from the IDE.

TIP

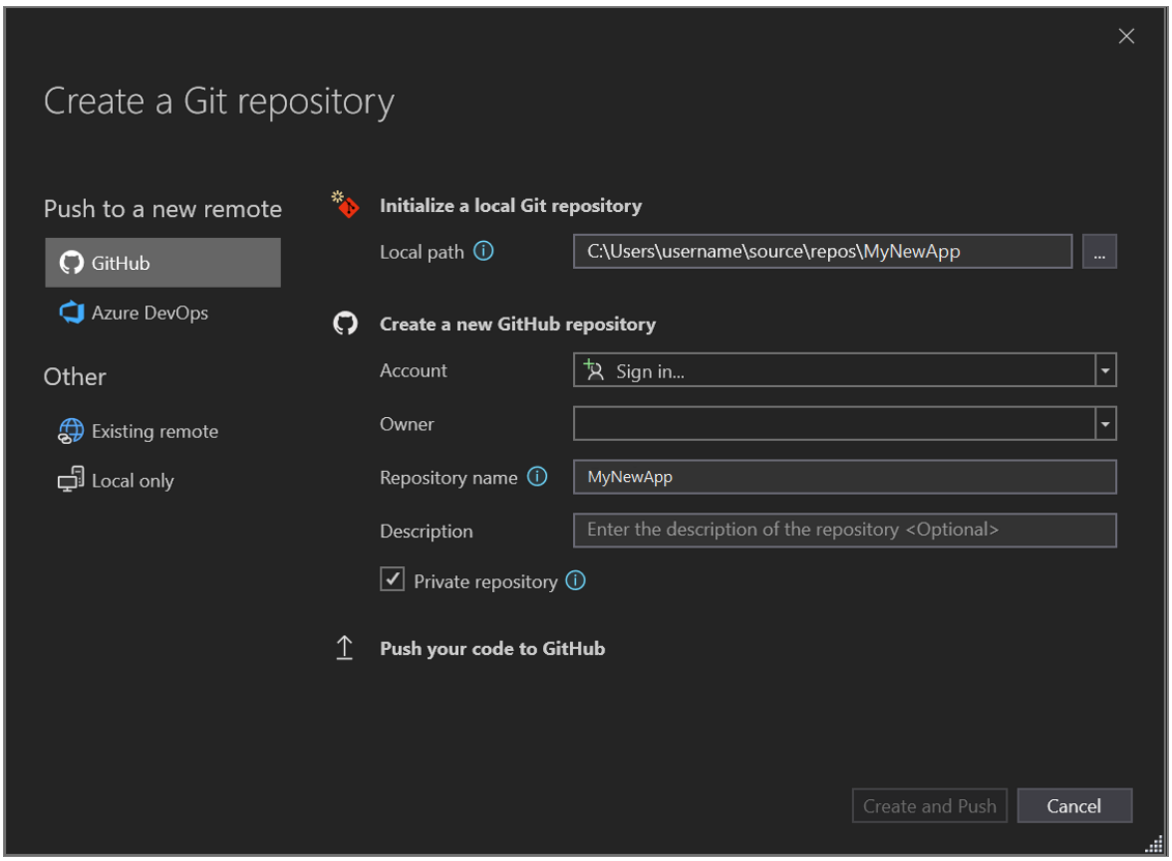
Git is the most widely used modern version control system, so whether you're a professional developer or you're learning how to code, Git can be very useful. If you're new to Git, the https://git-scm.com/ website is a good place to start. There, you can find cheat sheets, a popular online book, and Git Basics videos.

To associate your code with Git, you start by creating a new Git repository where your code is located. Here's how:

1. In the status bar at the bottom-right corner of Visual Studio, select **Add to Source Control**, and then select **Git**.



2. In the **Create a Git repository** dialog box, sign in to GitHub.

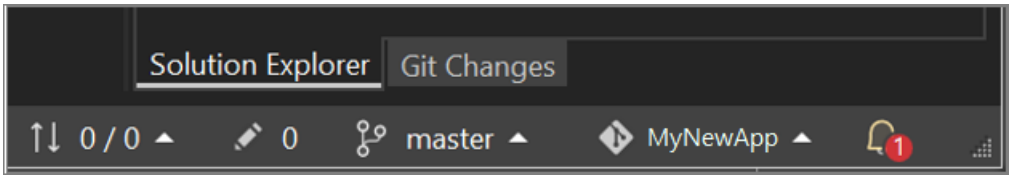


The repository name auto-populates based on your folder location. By default, your new repository is private, which means you're the only one who can access it.

> **TIP**
> Whether your repository is public or private, it's best to have a remote backup of your code stored securely on GitHub. Even if you aren't working with a team, a remote repository makes your code available to you from any computer.

3. Select **Create and Push**.

After you create your repository, you see status details in the status bar.



The first icon with the arrows shows how many outgoing/incoming commits are in your current branch. You can use this icon to pull any incoming commits or push any outgoing commits. You can also choose to view these commits first. To do so, select the icon, and then select **View Outgoing/Incoming**.

The second icon with the pencil shows the number of uncommitted changes to your code. You can select this icon to view those changes in the **Git Changes** window.

To learn more about how to use Git with your app, see the Visual Studio version control documentation.

The finished app

Congratulations! You've completed the code for the calculator app, built and debugged it, and added it to a repo, all in Visual Studio.

Next steps

Learn more about Visual Studio for C++

The usual starting point for a C++ programmer is a "Hello, world!" application that runs on the command line. That's what you'll create in Visual Studio in this article, and then we'll move on to something more challenging: a calculator app.

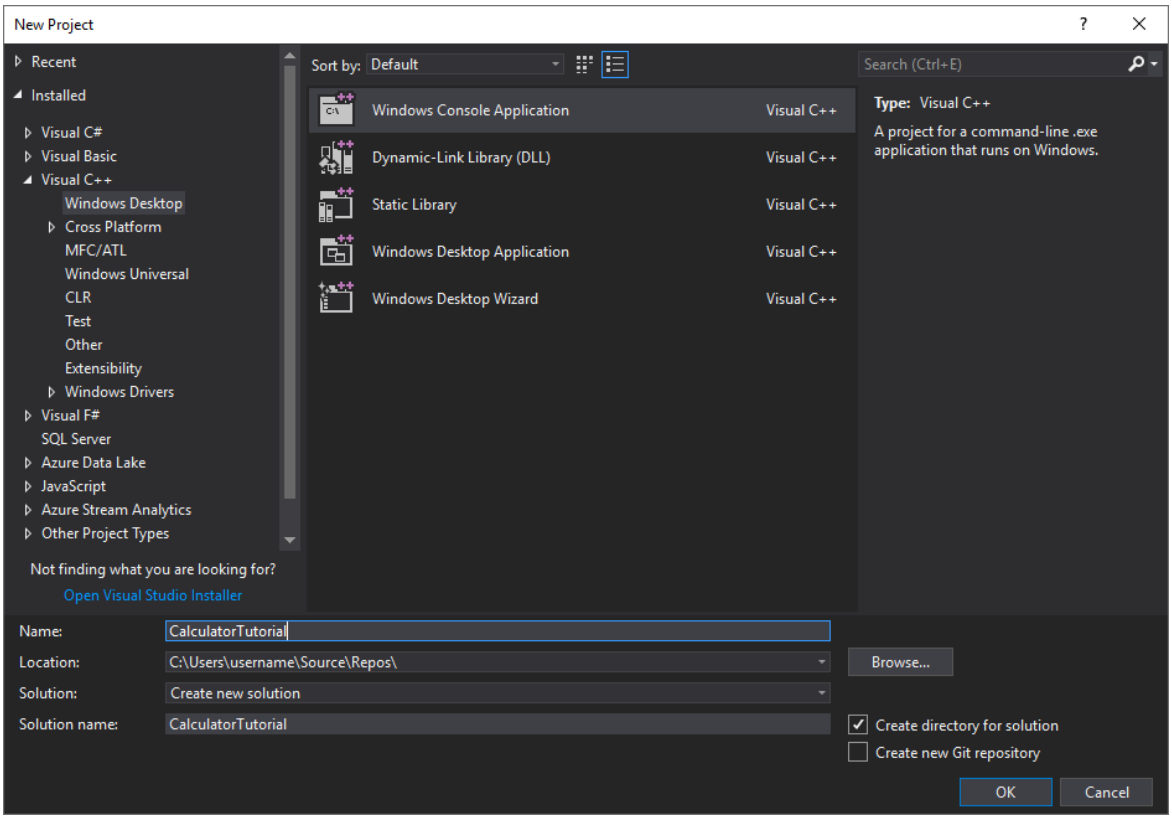
Prerequisites

- Have Visual Studio with the **Desktop development with C++** workload installed and running on your computer. If it's not installed yet, see Install C++ support in Visual Studio.

Create your app project

Visual Studio uses *projects* to organize the code for an app, and *solutions* to organize your projects. A project contains all the options, configurations, and rules used to build your apps. It also manages the relationship between all the project's files and any external files. To create your app, first, you'll create a new project and solution.

1. On the menubar in Visual Studio, choose **File** > **New** > **Project**. The **New Project** window opens.
2. On the left sidebar, make sure **Visual C++** is selected. In the center, choose **Windows Console Application**.
3. In the **Name** edit box at the bottom, name the new project *CalculatorTutorial*, then choose **OK**.



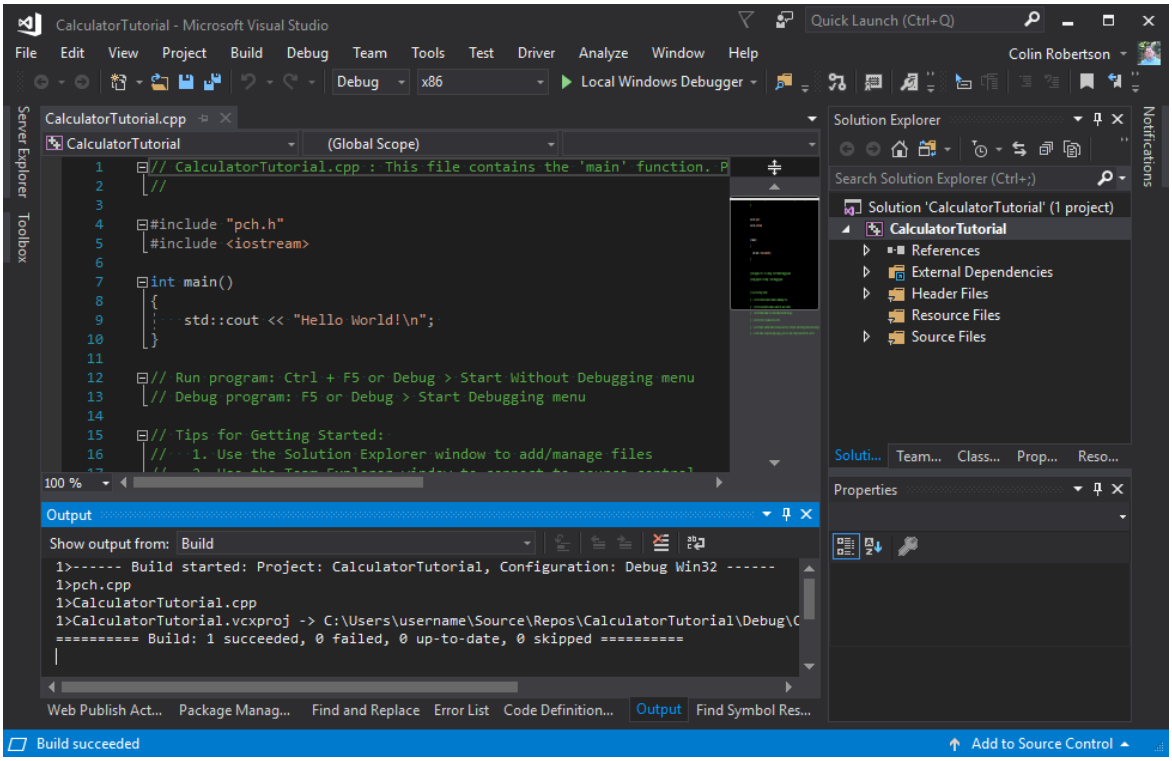
An empty C++ Windows console application gets created. Console applications use a Windows console window to display output and accept user input. In Visual Studio, an editor window opens and shows the generated code:

```
// CalculatorTutorial.cpp : This file contains the 'main' function. Program execution begins and ends
there.
//

#include "pch.h"
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu

// Tips for Getting Started:
//   1. Use the Solution Explorer window to add/manage files
//   2. Use the Team Explorer window to connect to source control
//   3. Use the Output window to see build output and other messages
//   4. Use the Error List window to view errors
//   5. Go to Project > Add New Item to create new code files, or Project > Add Existing Item to add
existing code files to the project
//   6. In the future, to open this project again, go to File > Open > Project and select the .sln
file
```

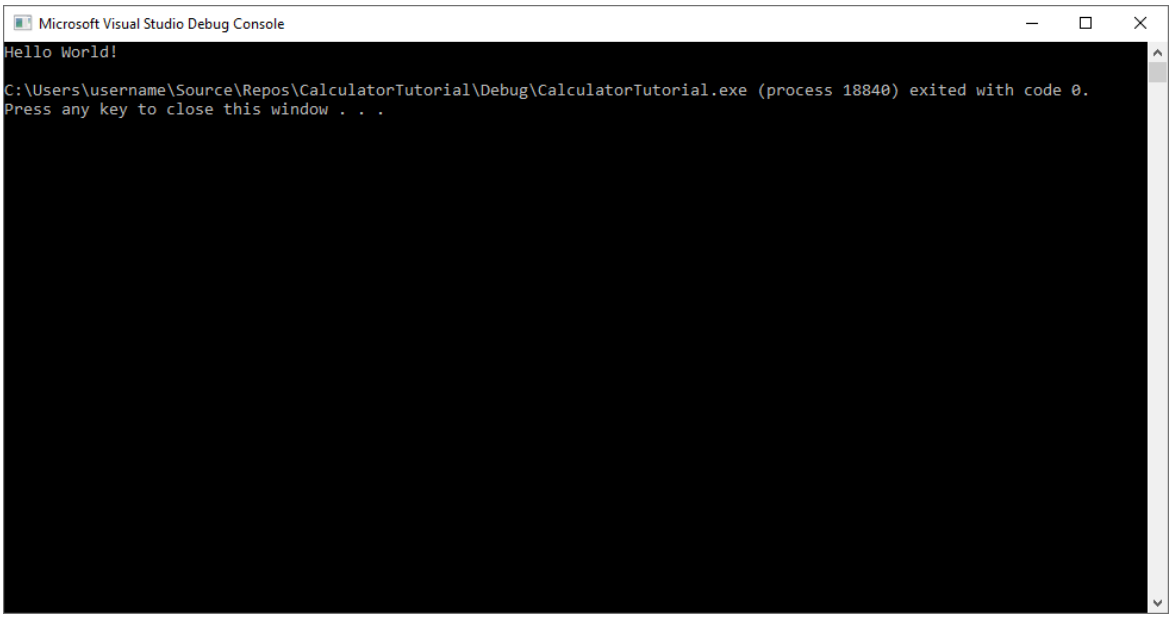
Verify that your new app builds and runs

The template for a new windows console application creates a simple C++ "Hello World" app. At this point, you can see how Visual Studio builds and runs the apps you create right from the IDE.

1. To build your project, choose **Build Solution** from the **Build** menu. The **Output** window shows the results of the build process.



2. To run the code, on the menu bar, choose **Debug**, **Start without debugging**.



A console window opens and then runs your app. When you start a console app in Visual Studio, it runs your code, then prints "Press any key to continue . . ." to give you a chance to see the output. Congratulations! You've created your first "Hello, world!" console app in Visual Studio!

3. Press a key to dismiss the console window and return to Visual Studio.

You now have the tools to build and run your app after every change, to verify that the code still works as you expect. Later, we'll show you how to debug it if it doesn't.

Edit the code

Now let's turn the code in this template into a calculator app.

1. In the *CalculatorTutorial.cpp* file, edit the code to match this example:

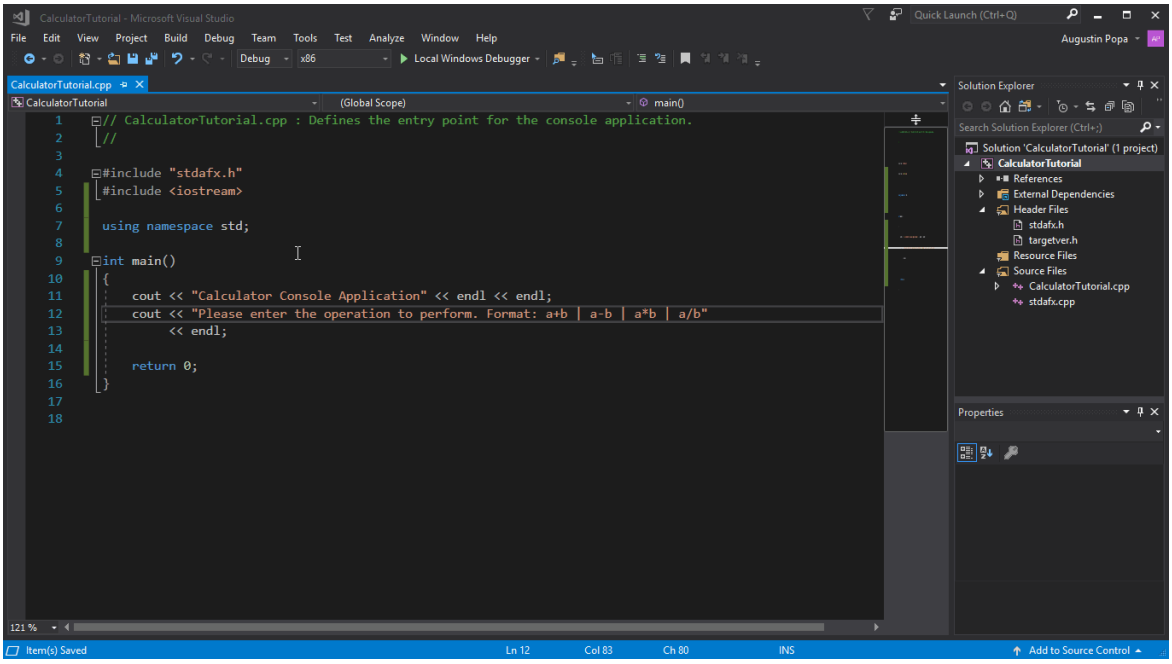
```
// CalculatorTutorial.cpp : This file contains the 'main' function. Program execution begins and ends
there.
//

#include "pch.h"
#include <iostream>

using namespace std;

int main()
{
    cout << "Calculator Console Application" << endl << endl;
    cout << "Please enter the operation to perform. Format: a+b | a-b | a*b | a/b"
        << endl;
    return 0;
}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu
// Tips for Getting Started:
//   1. Use the Solution Explorer window to add/manage files
//   2. Use the Team Explorer window to connect to source control
//   3. Use the Output window to see build output and other messages
//   4. Use the Error List window to view errors
//   5. Go to Project > Add New Item to create new code files, or Project > Add Existing Item to add
existing code files to the project
//   6. In the future, to open this project again, go to File > Open > Project and select the .sln
file
```

Understanding the code:

- The `#include` statements allow you to reference code located in other files. Sometimes, you may see a filename surrounded by angle brackets (< >); other times, it's surrounded by quotes (" "). In general, angle brackets are used when referencing the C++ Standard Library, while quotes are used for other files.
- The `#include "pch.h"` (or in Visual Studio 2017 and earlier, `#include "stdafx.h"`) line references something known as a precompiled header. These are often used by professional programmers to improve compilation times, but they are beyond the scope of this tutorial.
- The `using namespace std;` line tells the compiler to expect stuff from the C++ Standard Library to be used in this file. Without this line, each keyword from the library would have to be preceded with a `std::`, to denote its scope. For instance, without that line, each reference to `cout` would have to be written as `std::cout`. The `using` statement is added to make the code look more clean.
- The `cout` keyword is used to print to standard output in C++. The `<<` operator tells the compiler to send whatever is to the right of it to the standard output.
- The **endl** keyword is like the Enter key; it ends the line and moves the cursor to the next line. It is a better practice to put a `\n` inside the string (contained by "") to do the same thing, as `endl` always flushes the buffer and can hurt the performance of the program, but since this is a very small app, `endl` is used instead for better readability.
- All C++ statements must end with semicolons and all C++ applications must contain a `main()` function. This function is what the program runs at the start. All code must be accessible from `main()` in order to be used.

2. To save the file, enter **Ctrl+S**, or choose the **Save** icon near the top of the IDE, the floppy disk icon in the toolbar under the menu bar.

3. To run the application, press **Ctrl+F5** or go to the **Debug** menu and choose **Start Without Debugging**. If you get a pop-up that says **This project is out of date**, you may select **Do not show this dialog again**, and then choose **Yes** to build your application. You should see a console window appear that displays the text specified in the code.



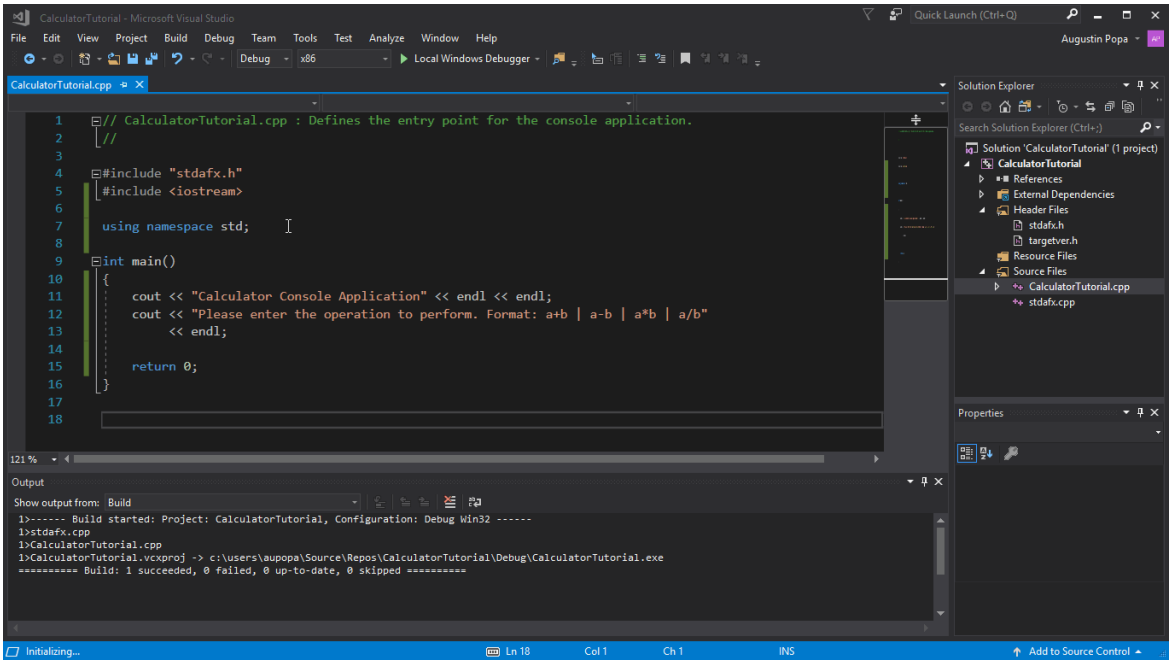
4. Close the console window when you're done.

Add code to do some math

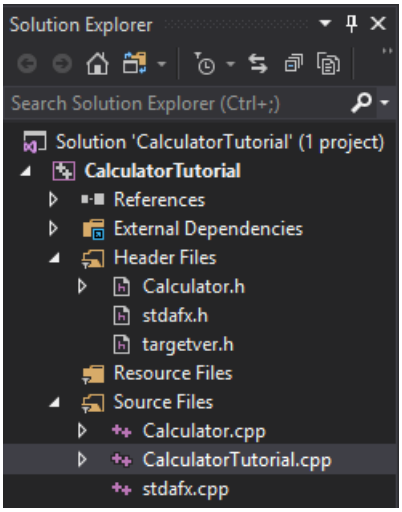
It's time to add some math logic.

To add a Calculator class

1. Go to the **Project** menu and choose **Add Class**. In the **Class Name** edit box, enter *Calculator*. Choose **OK**. Two new files get added to your project. To save all your changed files at once, press **Ctrl+Shift+S**. It's a keyboard shortcut for **File** > **Save All**. There's also a toolbar button for **Save All**, an icon of two floppy disks, found beside the **Save** button. In general, it's good practice to do **Save All** frequently, so you don't miss any files when you save.



A class is like a blueprint for an object that does something. In this case, we define a calculator and how it should work. The **Add Class** wizard you used above created .h and .cpp files that have the same name as the class. You can see a full list of your project files in the **Solution Explorer** window, visible on the side of the IDE. If the window isn't visible, you can open it from the menu bar: choose **View** > **Solution Explorer**.



You should now have three tabs open in the editor: *CalculatorTutorial.cpp*, *Calculator.h*, and *Calculator.cpp*. If you accidentally close one of them, you can reopen it by double-clicking it in the **Solution Explorer** window.

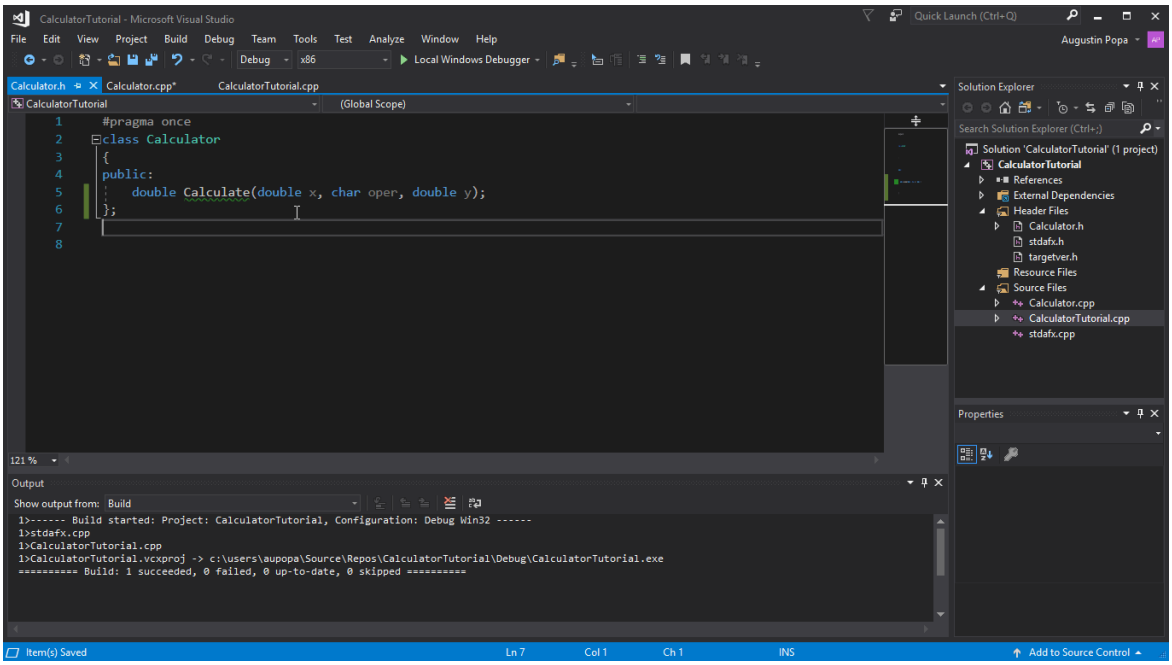
2. In **Calculator.h**, remove the `Calculator();` and `~Calculator();` lines that were generated, since you won't need them here. Next, add the following line of code so the file now looks like this:

```
#pragma once
class Calculator
{
public:
    double Calculate(double x, char oper, double y);
};
```

Understanding the code

- The line you added declares a new function called `Calculate`, which we'll use to run math operations for addition, subtraction, multiplication, and division.
- C++ code is organized into *header* (.h) files and *source* (.cpp) files. Several other file extensions are supported by various compilers, but these are the main ones to know about. Functions and variables are normally *declared*, that is, given a name and a type, in header files, and *implemented*, or given a definition, in source files. To access code defined in another file, you can use `#include "filename.h"`, where 'filename.h' is the name of the file that declares the variables or functions you want to use.
- The two lines you deleted declared a *constructor* and *destructor* for the class. For a simple class like this one, the compiler creates them for you, and their uses are beyond the scope of this tutorial.
- It's good practice to organize your code into different files based on what it does, so it's easy to find the code you need later. In our case, we define the `Calculator` class separately from the file containing the `main()` function, but we plan to reference the `Calculator` class in `main()`.

3. You'll see a green squiggle appear under `Calculate`. It's because we haven't defined the `Calculate` function in the .cpp file. Hover over the word, click the lightbulb that pops up, and choose **Create definition of 'Calculate' in Calculator.cpp**. A pop-up appears that gives you a peek of the code change that was made in the other file. The code was added to *Calculator.cpp*.



Currently, it just returns 0.0. Let's change that. Press **Esc** to close the pop-up.

4. Switch to the *Calculator.cpp* file in the editor window. Remove the `Calculator()` and `~Calculator()` sections (as you did in the .h file) and add the following code to `Calculate()`:

```
#include "pch.h"
#include "Calculator.h"

double Calculator::Calculate(double x, char oper, double y)
{
    switch(oper)
    {
        case '+':
            return x + y;
        case '-':
            return x - y;
        case '*':
            return x * y;
        case '/':
            return x / y;
        default:
            return 0.0;
    }
}
```

Understanding the code

- The function `Calculate` consumes a number, an operator, and a second number, then performs the requested operation on the numbers.
- The switch statement checks which operator was provided, and only executes the case corresponding to that operation. The default: case is a fallback in case the user types an operator that isn't accepted, so the program doesn't break. In general, it's best to handle invalid user input in a more elegant way, but this is beyond the scope of this tutorial.
- The `double` keyword denotes a type of number that supports decimals. This way, the calculator can handle both decimal math and integer math. The `Calculate` function is required to always return such a number due to the `double` at the very start of the code (this denotes the function's return type), which is why we return 0.0 even in the default case.
- The .h file declares the function *prototype*, which tells the compiler upfront what parameters it requires, and what return type to expect from it. The .cpp file has all the implementation details of the function.

If you build and run the code again at this point, it will still exit after asking which operation to perform. Next, you'll modify the `main` function to do some calculations.

To call the Calculator class member functions

1. Now let's update the `main` function in *CalculatorTutorial.cpp*:

```
// CalculatorTutorial.cpp : This file contains the 'main' function. Program execution begins and ends
there.
//

#include "pch.h"
#include <iostream>
#include "Calculator.h"

using namespace std;

int main()
{
    double x = 0.0;
    double y = 0.0;
    double result = 0.0;
    char oper = '+';

    cout << "Calculator Console Application" << endl << endl;
    cout << "Please enter the operation to perform. Format: a+b | a-b | a*b | a/b"
         << endl;

    Calculator c;
    while (true)
    {
        cin >> x >> oper >> y;
        result = c.Calculate(x, oper, y);
        cout << "Result is: " << result << endl;
    }

    return 0;
}
```

Understanding the code

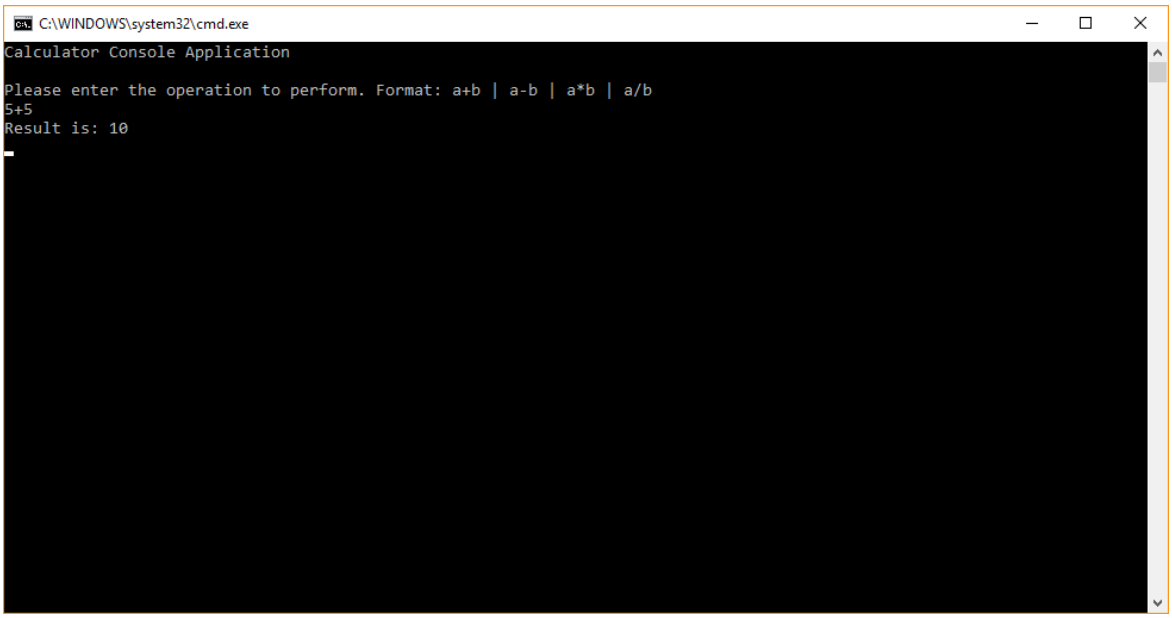
- Since C++ programs always start at the `main()` function, we need to call our other code from there, so a `#include` statement is needed.
- Some initial variables `x`, `y`, `oper`, and `result` are declared to store the first number, second number, operator, and final result, respectively. It is always good practice to give them some initial values to avoid undefined behavior, which is what is done here.
- The `Calculator c;` line declares an object named 'c' as an instance of the `Calculator` class. The class itself is just a blueprint for how calculators work; the object is the specific calculator that does the math.
- The `while (true)` statement is a loop. The code inside the loop continues to execute over and over again as long as the condition inside the `()` holds true. Since the condition is simply listed as `true`, it's always true, so the loop runs forever. To close the program, the user must manually close the console window. Otherwise, the program always waits for new input.
- The `cin` keyword is used to accept input from the user. This input stream is smart enough to process a line of text entered in the console window and place it inside each of the variables listed, in order, assuming the user input matches the required specification. You can modify this line to accept different types of input, for instance, more than two numbers, though the `Calculate()` function would also need to be updated to handle this.
- The `c.Calculate(x, oper, y);` expression calls the `Calculate` function defined earlier, and supplies the entered input values. The function then returns a number that gets stored in `result`.
- Finally, `result` is printed to the console, so the user sees the result of the calculation.

Build and test the code again

Now it's time to test the program again to make sure everything works properly.

1. Press **Ctrl+F5** to rebuild and start the app.

2. Enter `5 + 5`, and press **Enter**. Verify that the result is 10.

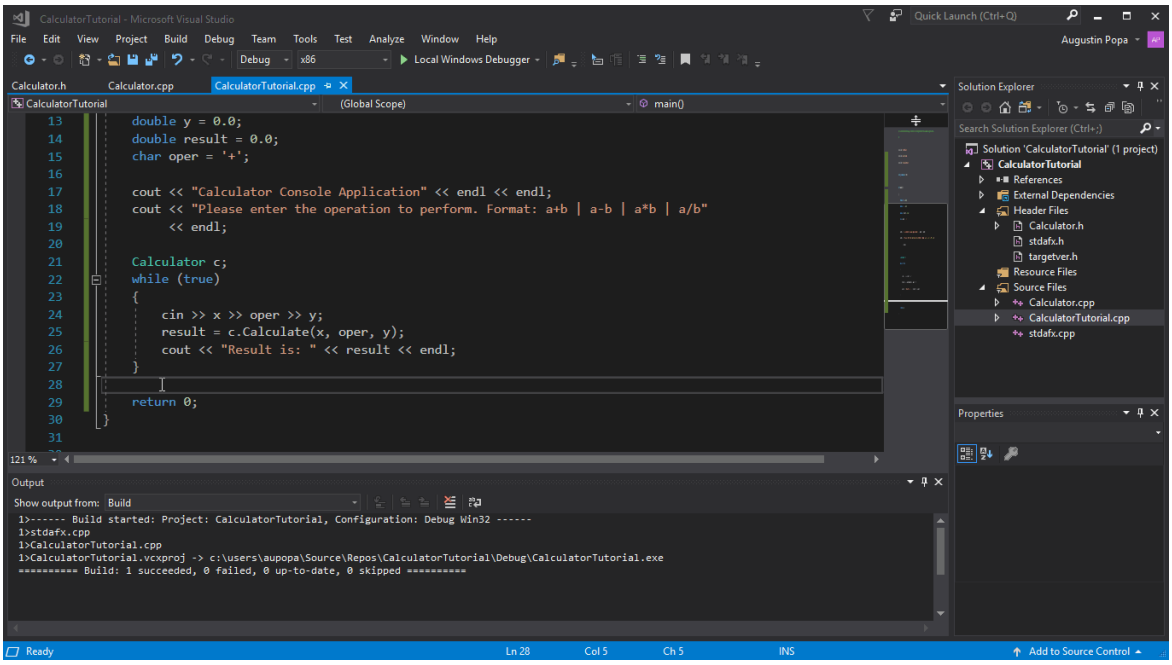


Debug the app

Since the user is free to type anything into the console window, let's make sure the calculator handles some input as expected. Instead of running the program, let's debug it instead, so we can inspect what it's doing in detail, step-by-step.

To run the app in the debugger

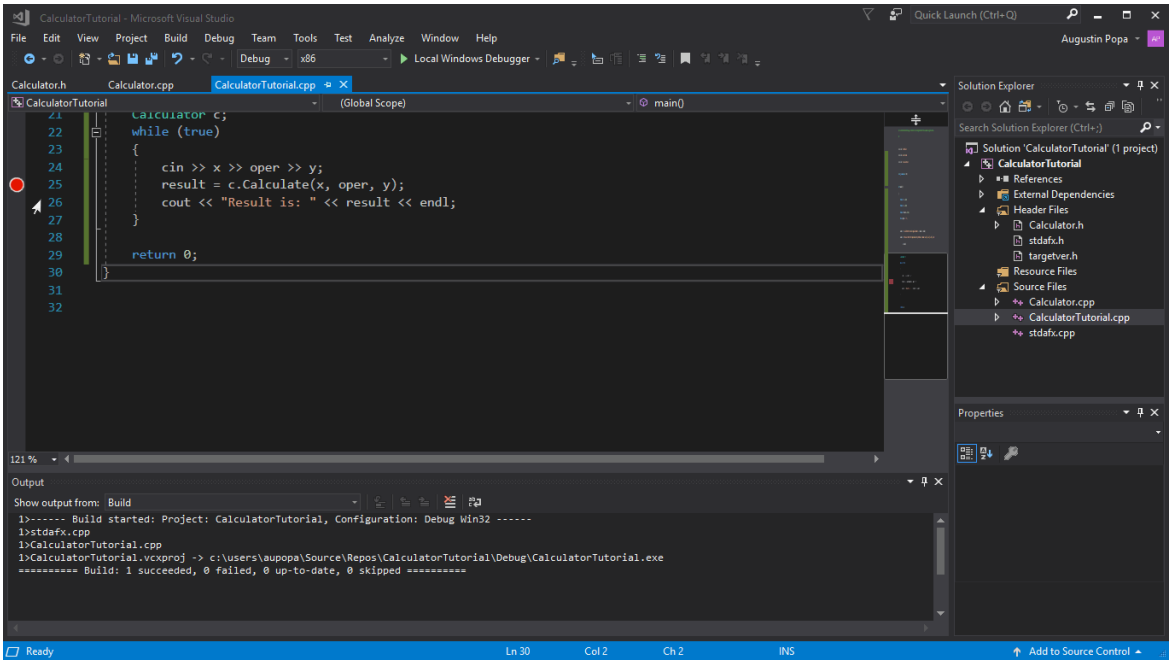
1. Set a breakpoint on the `result = c.Calculate(x, oper, y);` line, just after the user was asked for input. To set the breakpoint, click next to the line in the gray vertical bar along the left edge of the editor window. A red dot appears.



   Now when we debug the program, it always pauses execution at that line. We already have a rough idea that the program works for simple cases. Since we don't want to pause execution every time, let's make the breakpoint conditional.

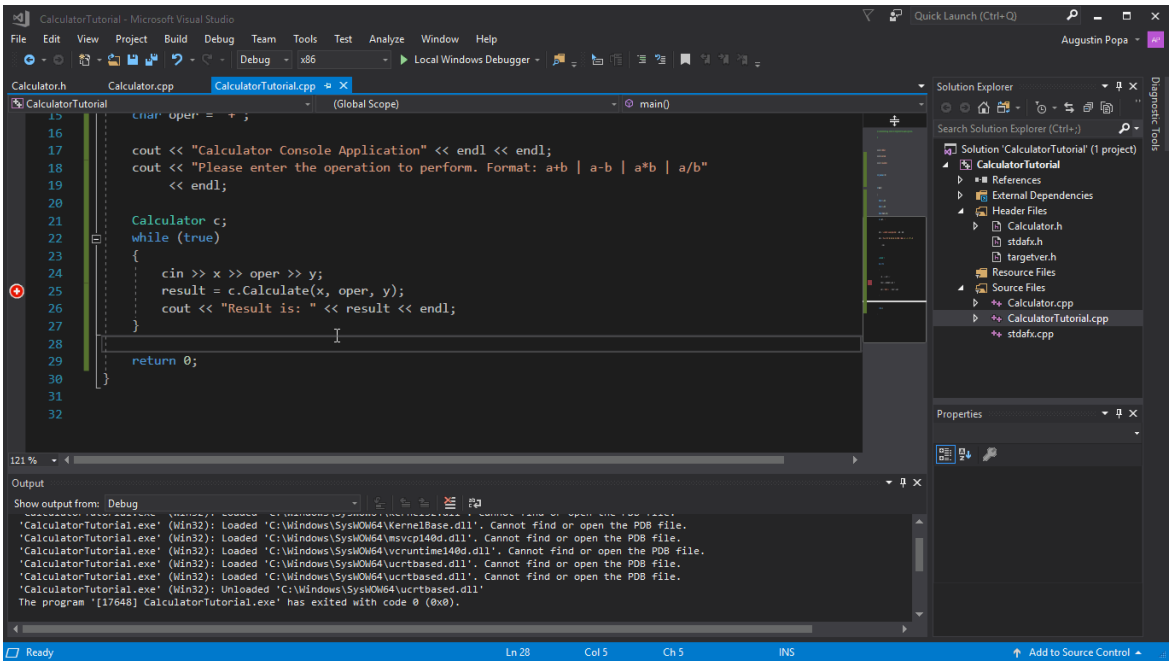
2. Right-click the red dot that represents the breakpoint, and choose **Conditions**. In the edit box for the

condition, enter `(y == 0) && (oper == '/')`. Choose the **Close** button when you're done. The condition is saved automatically.



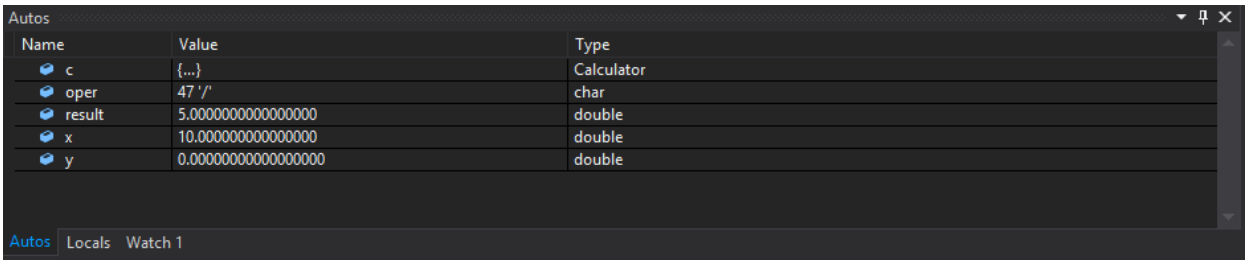
Now we pause execution at the breakpoint specifically if a division by 0 is attempted.

3. To debug the program, press **F5**, or choose the **Local Windows Debugger** toolbar button that has the green arrow icon. In your console app, if you enter something like "5 - 0", the program behaves normally and keeps running. However, if you type "10 / 0", it pauses at the breakpoint. You can even put any number of spaces between the operator and numbers; `cin` is smart enough to parse the input appropriately.

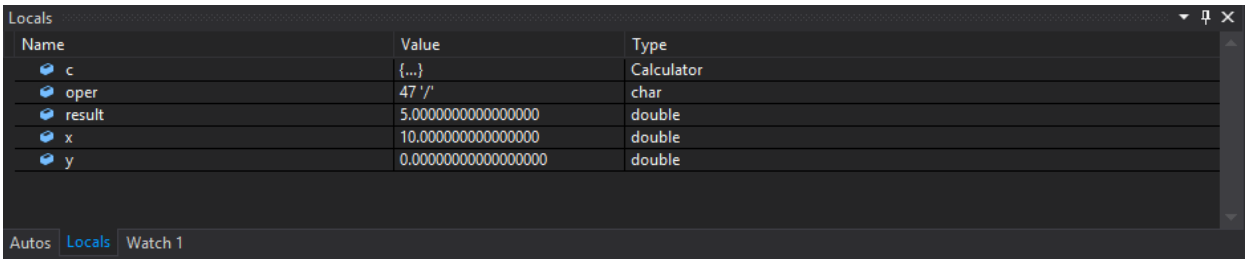


Useful windows in the debugger

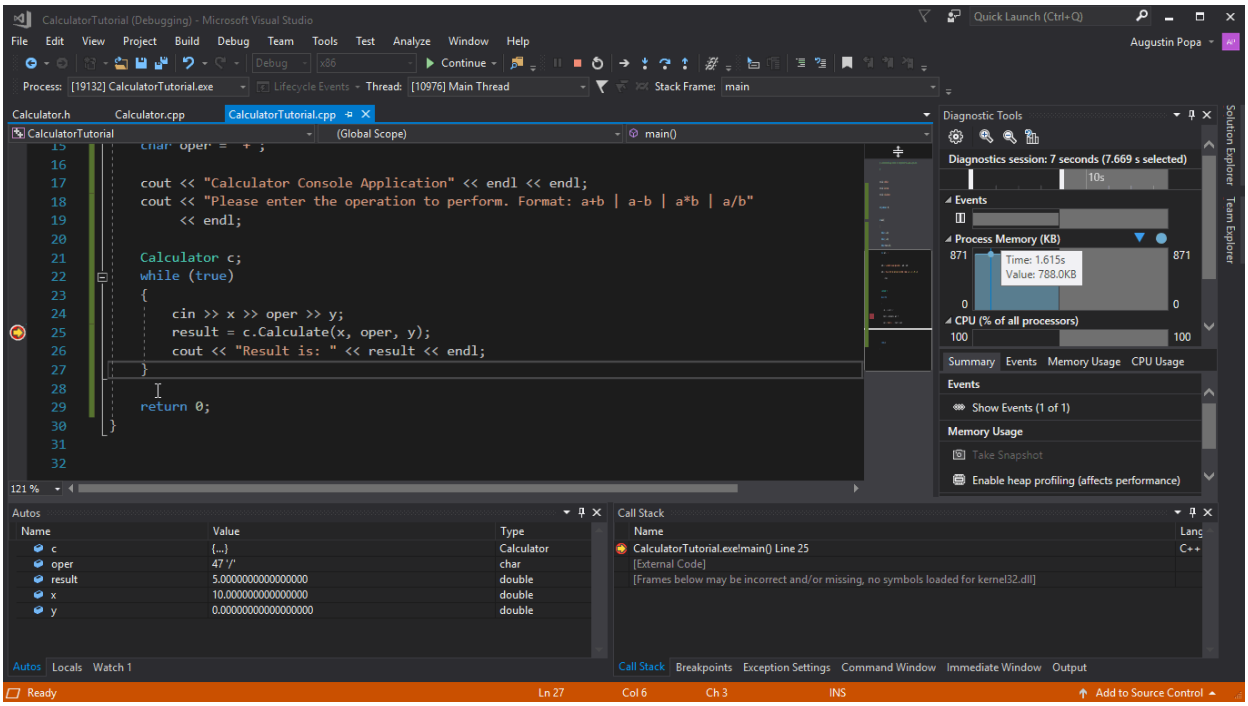
Whenever you debug your code, you may notice that some new windows appear. These windows can assist your debugging experience. Take a look at the **Autos** window. The **Autos** window shows you the current values of variables used at least three lines before and up to the current line.



To see all of the variables from that function, switch to the **Locals** window. You can actually modify the values of these variables while debugging, to see what effect they would have on the program. In this case, we'll leave them alone.



You can also just hover over variables in the code itself to see their current values where the execution is currently paused. Make sure the editor window is in focus by clicking on it first.

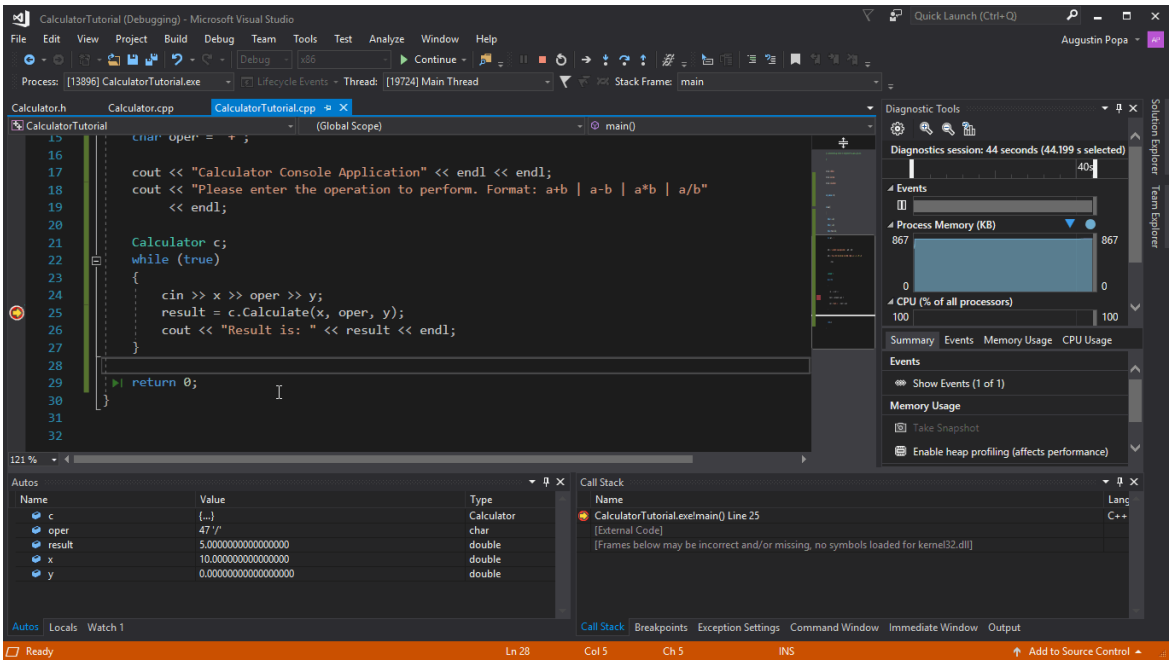


To continue debugging

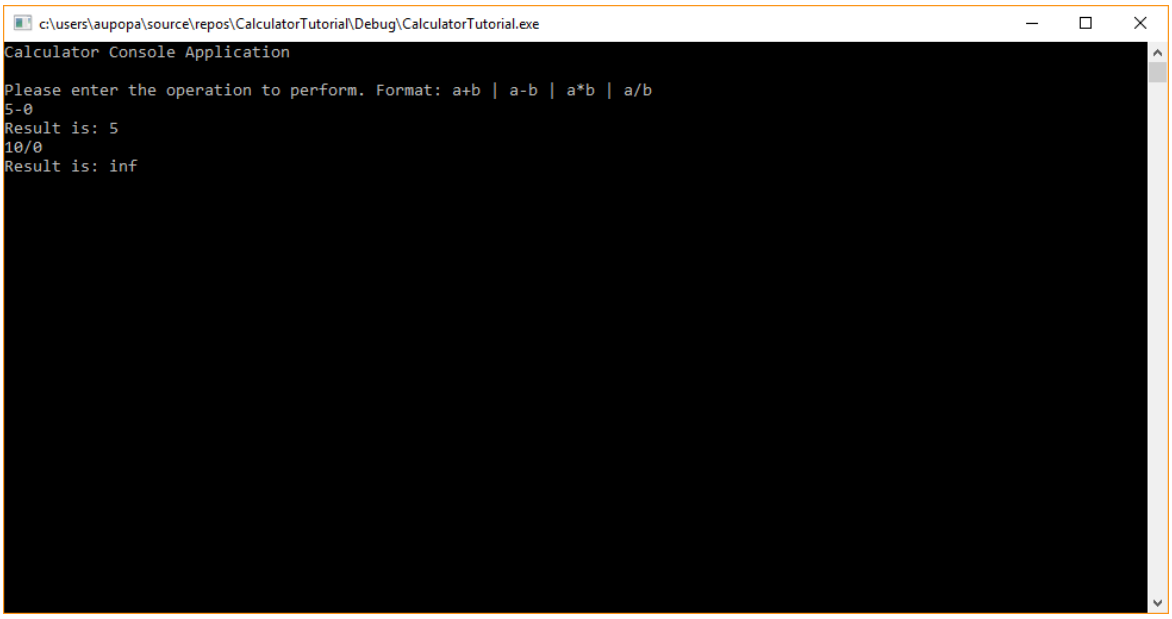
1. The yellow line on the left shows the current point of execution. The current line calls `Calculate`, so press **F11** to **Step Into** the function. You'll find yourself in the body of the `Calculate` function. Be careful with **Step Into**; if you do it too much, you may waste a lot of time. It goes into any code you use on the line you are on, including standard library functions.

2. Now that the point of execution is at the start of the `Calculate` function, press **F10** to move to the next line in the program's execution. **F10** is also known as **Step Over**. You can use **Step Over** to move from line to line, without delving into the details of what is occurring in each part of the line. In general you should use **Step Over** instead of **Step Into**, unless you want to dive more deeply into code that is being called from elsewhere (as you did to reach the body of `Calculate`).

3. Continue using **F10** to **Step Over** each line until you get back to the `main()` function in the other file, and stop on the `cout` line.



It looks like the program is doing what is expected: it takes the first number, and divides it by the second. On the `cout` line, hover over the `result` variable or take a look at `result` in the **Autos** window. You'll see its value is listed as "inf", which doesn't look right, so let's fix it. The `cout` line just outputs whatever value is stored in `result`, so when you step one more line forward using **F10**, the console window displays:



This result happens because division by zero is undefined, so the program doesn't have a numerical answer to the requested operation.

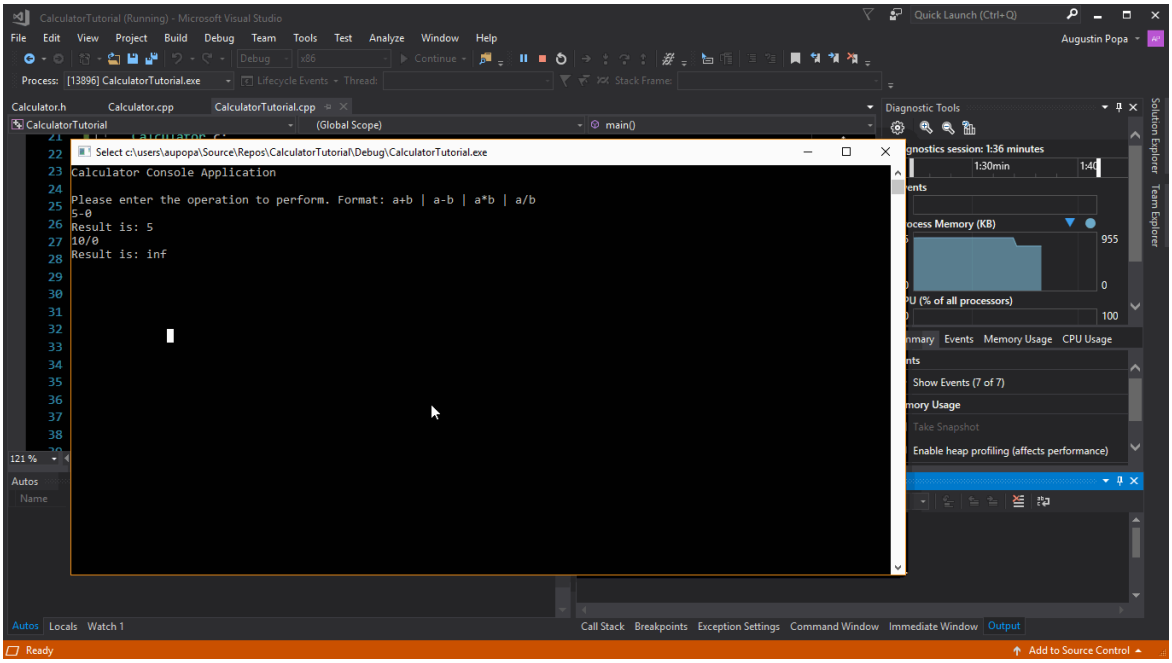
To fix the "divide by zero" error

Let's handle division by zero more gracefully, so a user can understand the problem.

1. Make the following changes in *CalculatorTutorial.cpp*. (You can leave the program running as you edit, thanks to a debugger feature called **Edit and Continue**):

```
// CalculatorTutorial.cpp : This file contains the 'main' function. Program execution begins and ends
there.
//

#include "pch.h"
#include <iostream>
#include "Calculator.h"

using namespace std;

int main()
{
    double x = 0.0;
    double y = 0.0;
    double result = 0.0;
    char oper = '+';

    cout << "Calculator Console Application" << endl << endl;
    cout << "Please enter the operation to perform. Format: a+b | a-b | a*b | a/b" << endl;

    Calculator c;
    while (true)
    {
        cin  >> x  >> oper  >> y;
        if (oper == '/' && y == 0)
        {
            cout << "Division by 0 exception" << endl;
            continue;
        }
        else
        {
            result = c.Calculate(x, oper, y);
        }
        cout << "Result is: " << result << endl;
    }

    return 0;
}
```

2. Now press **F5** once. Program execution continues all the way until it has to pause to ask for user input. Enter `10 / 0` again. Now, a more helpful message is printed. The user is asked for more input, and the program continues executing normally.



NOTE

When you edit code while in debugging mode, there is a risk of code becoming stale. This happens when the debugger is still running your old code, and has not yet updated it with your changes. The debugger pops up a dialog to inform you when this happens. Sometimes, you may need to press **F5** to refresh the code being executed. In particular, if you make a change inside a function while the point of execution is inside that function, you'll need to step out of the function, then back into it again to get the updated code. If that doesn't work for some reason and you see an error message, you can stop debugging by clicking on the red square in the toolbar under the menus at the top of the IDE, then start debugging again by entering **F5** or by choosing the green "play" arrow beside the stop button on the toolbar.

Understanding the Run and Debug shortcuts

- **F5** (or **Debug** > **Start Debugging**) starts a debugging session if one isn't already active, and runs the program until a breakpoint is hit or the program needs user input. If no user input is needed and no breakpoint is available to hit, the program terminates and the console window closes itself when the program finishes running. If you have something like a "Hello World" program to run, use **Ctrl+F5** or set a breakpoint before you enter **F5** to keep the window open.
- **Ctrl+F5** (or **Debug** > **Start Without Debugging**) runs the application without going into debug mode. This is slightly faster than debugging, and the console window stays open after the program finishes executing.
- **F10** (known as **Step Over**) lets you iterate through code, line-by-line, and visualize how the code is run and what variable values are at each step of execution.
- **F11** (known as **Step Into**) works similarly to **Step Over**, except it steps into any functions called on the line of execution. For example, if the line being executed calls a function, pressing **F11** moves the pointer into the body of the function, so you can follow the function's code being run before coming back to the line you started at. Pressing **F10** steps over the function call and just moves to the next line; the function call still happens, but the program doesn't pause to show you what it's doing.

Close the app

- If it's still running, close the console window for the calculator app.

Congratulations! You've completed the code for the calculator app, and built and debugged it in Visual Studio.

Next steps

Learn more about Visual Studio for C++

Create a console calculator in C++

9/2/2022 • 39 minutes to read • Edit Online

The usual starting point for a C++ programmer is a "Hello, world!" application that runs on the command line. That's what you'll create first in Visual Studio in this article, and then we'll move on to something more challenging: a calculator app.

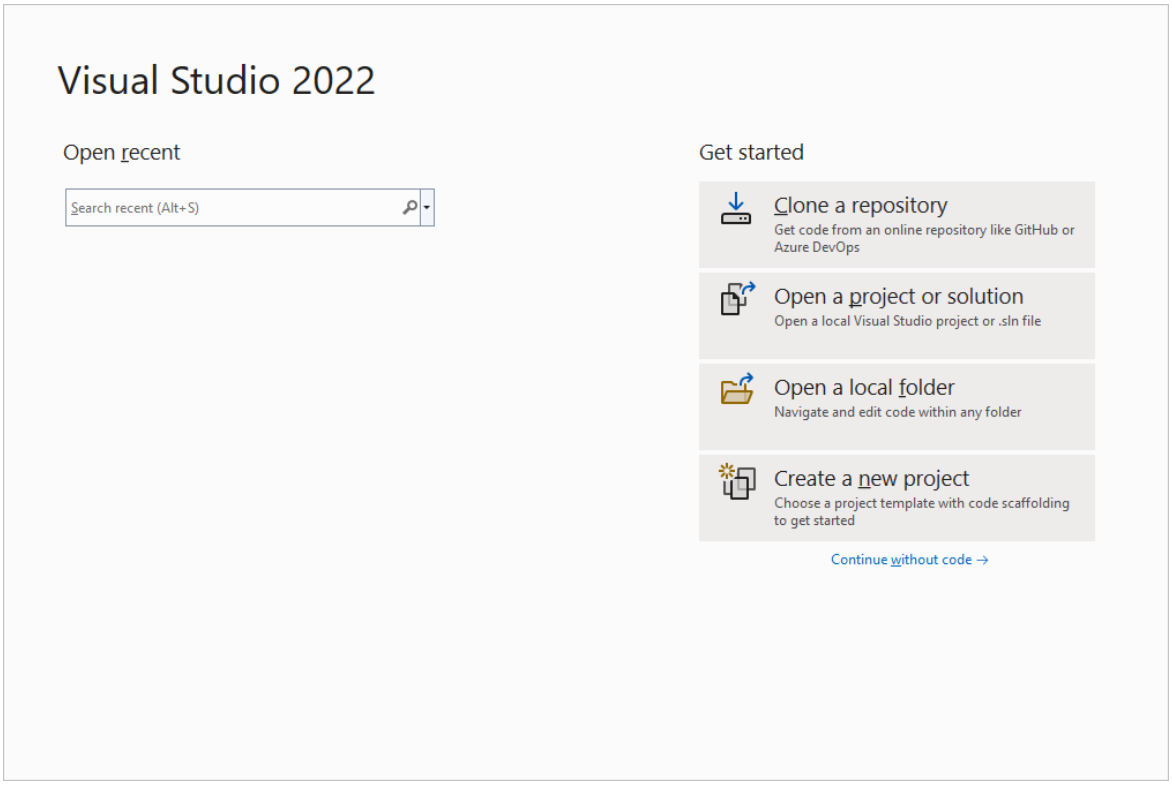
Prerequisites

- Have Visual Studio with the **Desktop development with C++** workload installed and running on your computer. If it's not installed yet, see Install C++ support in Visual Studio.

Create your app project

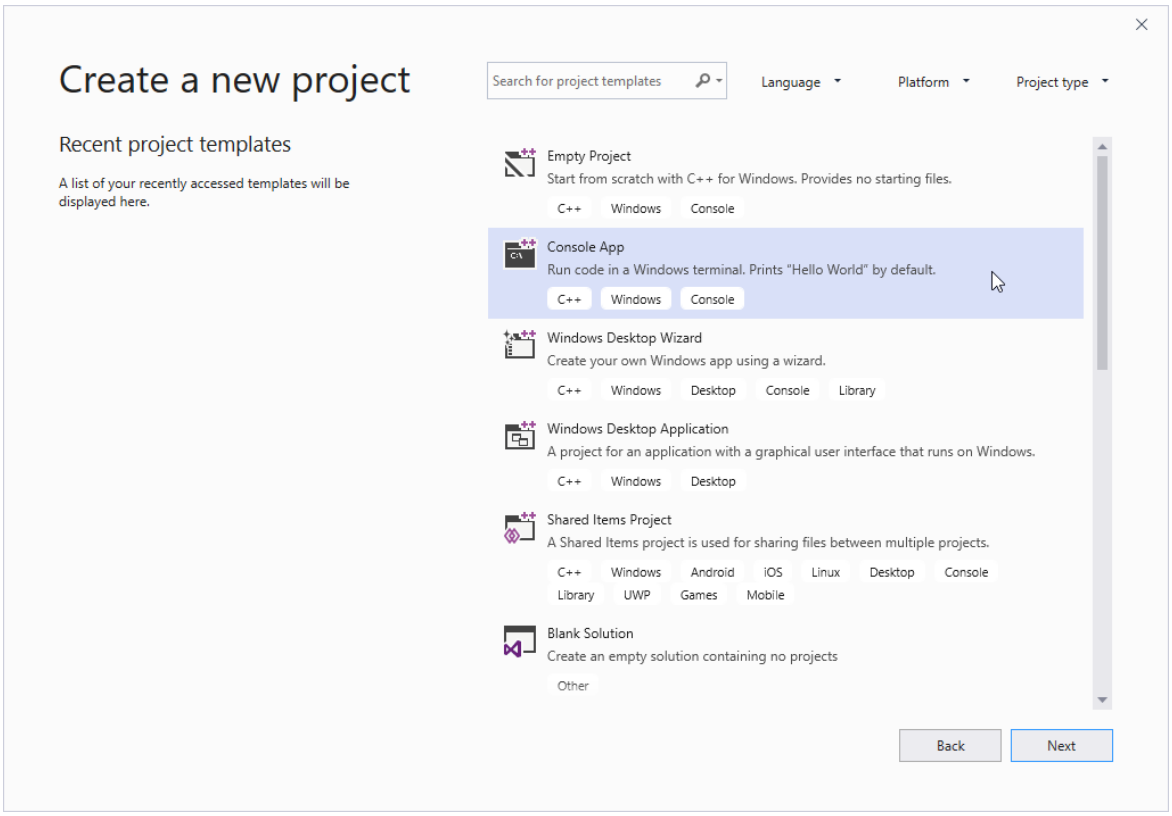
Visual Studio uses *projects* to organize the code for an app, and *solutions* to organize your projects. A project contains all the options, configurations, and rules used to build your apps. It also manages the relationship between all the project's files and any external files. To create your app, first, you'll create a new project and solution.

1. If you've just started Visual Studio, you'll see the Visual Studio Start dialog box. Choose **Create a new project** to get started.



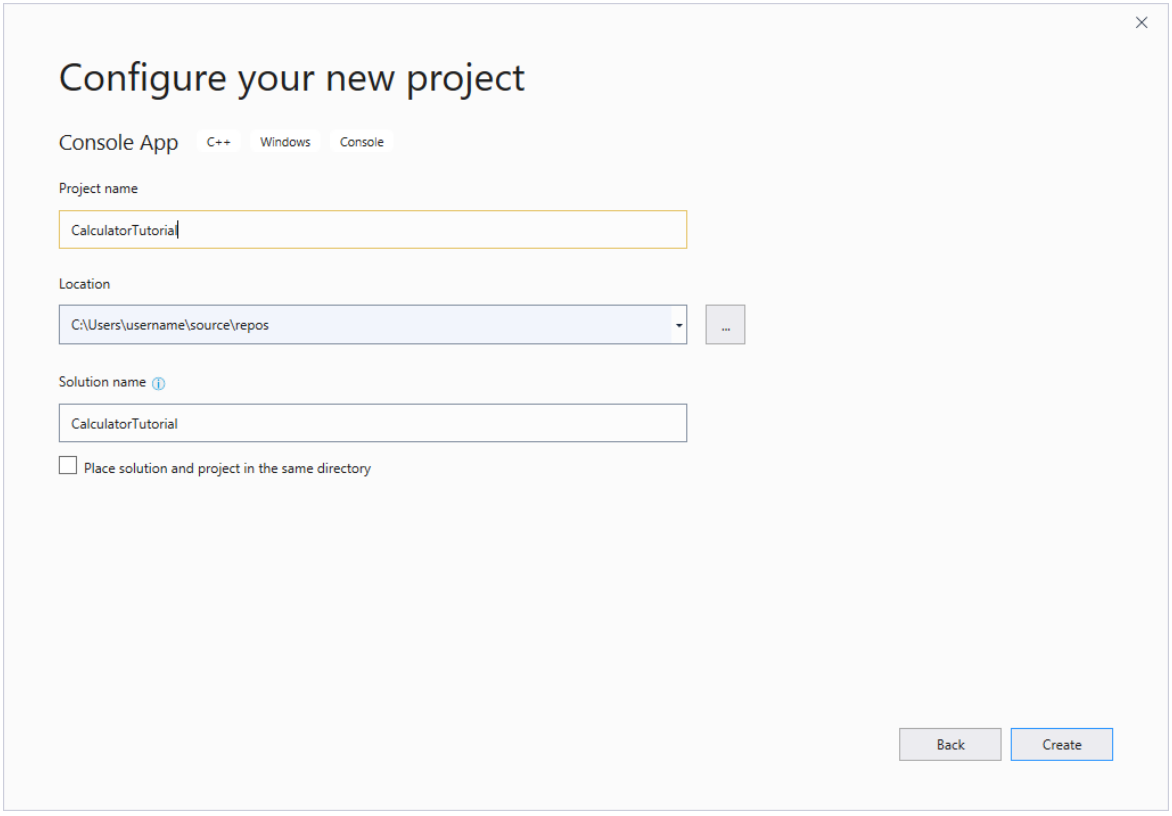
   Otherwise, on the menubar in Visual Studio, choose **File** > **New** > **Project**. The **Create a new project** window opens.

2. In the list of project templates, choose **Console App**, then choose **Next**.



> **IMPORTANT**
>
> Make sure you choose the C++ version of the **Console App** template. It has the **C++**, **Windows**, and **Console** tags, and the icon has "++" in the corner.

3. In the **Configure your new project** dialog box, select the **Project name** edit box, name your new project *CalculatorTutorial*, then choose **Create**.



An empty C++ Windows console application gets created. Console applications use a Windows console window to display output and accept user input. In Visual Studio, an editor window opens and shows the

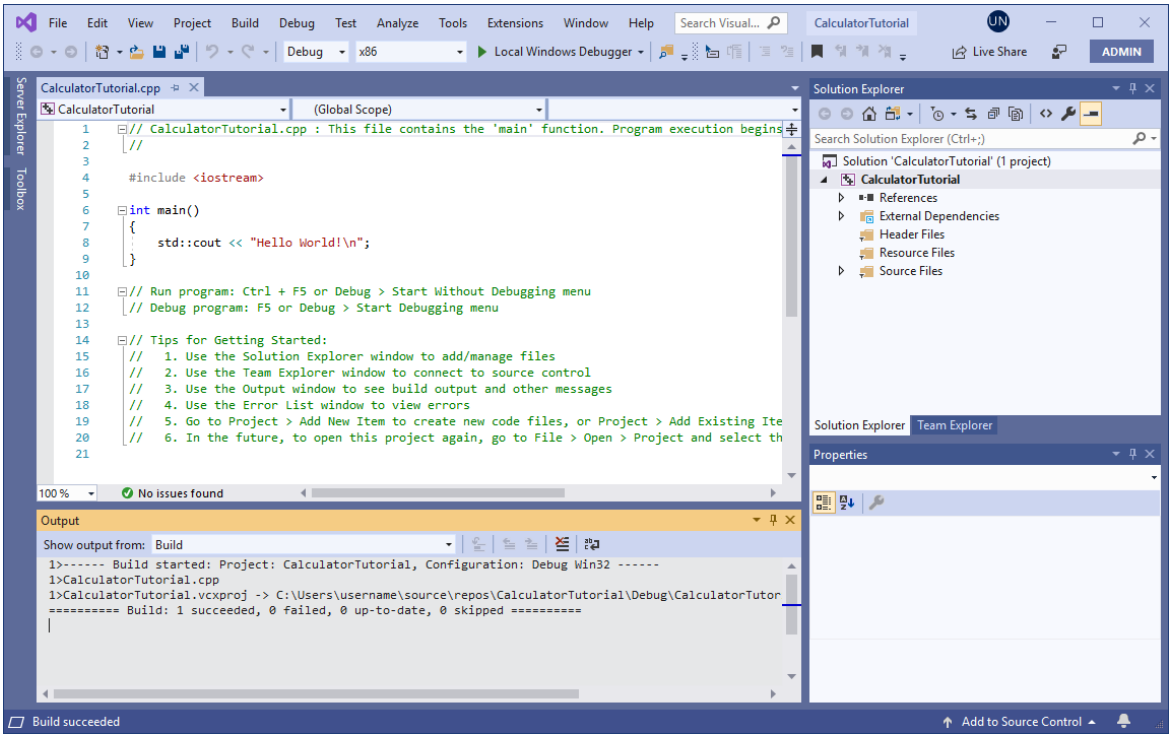
generated code:

```
// CalculatorTutorial.cpp : This file contains the 'main' function. Program execution begins and ends
there.
//

#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu

// Tips for Getting Started:
//   1. Use the Solution Explorer window to add/manage files
//   2. Use the Team Explorer window to connect to source control
//   3. Use the Output window to see build output and other messages
//   4. Use the Error List window to view errors
//   5. Go to Project > Add New Item to create new code files, or Project > Add Existing Item to add
existing code files to the project
//   6. In the future, to open this project again, go to File > Open > Project and select the .sln
file
```

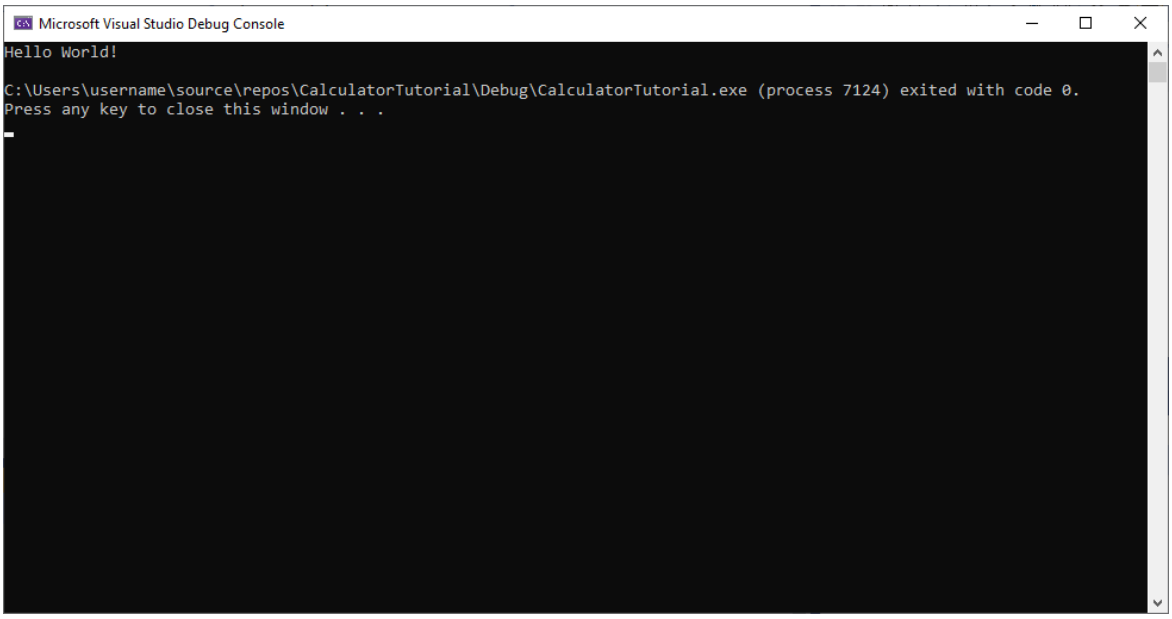
Verify that your new app builds and runs

The template for a new Windows console application creates a simple C++ "Hello World" app. At this point, you can see how Visual Studio builds and runs the apps you create right from the IDE.

1. To build your project, choose **Build Solution** from the **Build** menu. The **Output** window shows the results of the build process.



2. To run the code, on the menu bar, choose **Debug**, **Start without debugging**.



A console window opens and then runs your app. When you start a console app in Visual Studio, it runs your code, then prints "Press any key to close this window . . ." to give you a chance to see the output. Congratulations! You've created your first "Hello, world!" console app in Visual Studio!

3. Press a key to dismiss the console window and return to Visual Studio.

You now have the tools to build and run your app after every change, to verify that the code still works as you expect. Later, we'll show you how to debug it if it doesn't.

Edit the code

Now let's turn the code in this template into a calculator app.

1. In the *CalculatorTutorial.cpp* file, edit the code to match this example:

```
// CalculatorTutorial.cpp : This file contains the 'main' function. Program execution begins and ends
there.
//

#include <iostream>

using namespace std;

int main()
{
    cout << "Calculator Console Application" << endl << endl;
    cout << "Please enter the operation to perform. Format: a+b | a-b | a*b | a/b"
        << endl;
    return 0;
}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu
// Tips for Getting Started:
//   1. Use the Solution Explorer window to add/manage files
//   2. Use the Team Explorer window to connect to source control
//   3. Use the Output window to see build output and other messages
//   4. Use the Error List window to view errors
//   5. Go to Project > Add New Item to create new code files, or Project > Add Existing Item to add
existing code files to the project
//   6. In the future, to open this project again, go to File > Open > Project and select the .sln
file
```

Understanding the code:

- The `#include` statements allow you to reference code located in other files. Sometimes, you may see a filename surrounded by angle brackets (< >); other times, it's surrounded by quotes (" "). In general, angle brackets are used when referencing the C++ Standard Library, while quotes are used for other files.
- The `using namespace std;` line tells the compiler to expect stuff from the C++ Standard Library to be used in this file. Without this line, each keyword from the library would have to be preceded with a `std::`, to denote its scope. For instance, without that line, each reference to `cout` would have to be written as `std::cout`. The `using` statement is added to make the code look more clean.
- The `cout` keyword is used to print to standard output in C++. The << operator tells the compiler to send whatever is to the right of it to the standard output.
- The **endl** keyword is like the Enter key; it ends the line and moves the cursor to the next line. It is a better practice to put a `\n` inside the string (contained by "") to do the same thing, as `endl` always flushes the buffer and can hurt the performance of the program, but since this is a very small app, `endl` is used instead for better readability.
- All C++ statements must end with semicolons and all C++ applications must contain a `main()` function. This function is what the program runs at the start. All code must be accessible from `main()` in order to be used.

2. To save the file, enter **Ctrl+S**, or choose the **Save** icon near the top of the IDE, the floppy disk icon in the toolbar under the menu bar.

3. To run the application, press **Ctrl+F5** or go to the **Debug** menu and choose **Start Without Debugging**. You should see a console window appear that displays the text specified in the code.

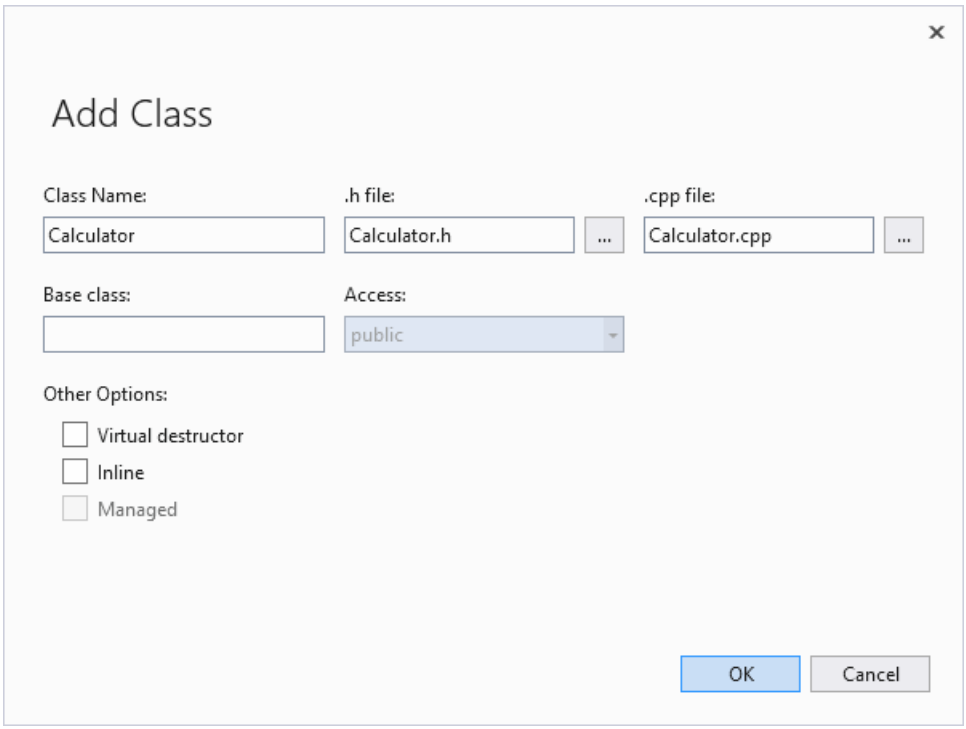
4. Close the console window when you're done.

Add code to do some math

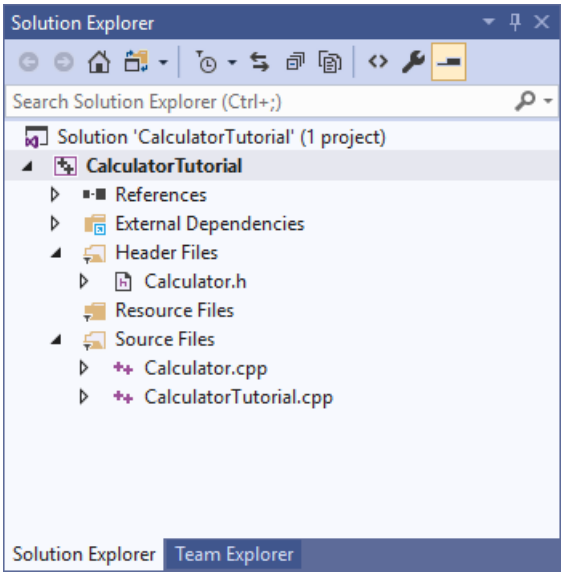
It's time to add some math logic.

To add a Calculator class

1. Go to the **Project** menu and choose **Add Class**. In the **Class Name** edit box, enter *Calculator*. Choose **OK**. Two new files get added to your project. To save all your changed files at once, press **Ctrl+Shift+S**. It's a keyboard shortcut for **File** > **Save All**. There's also a toolbar button for **Save All**, an icon of two floppy disks, found beside the **Save** button. In general, it's good practice to do **Save All** frequently, so you don't miss any files when you save.



A class is like a blueprint for an object that does something. In this case, we define a calculator and how it should work. The **Add Class** wizard you used above created .h and .cpp files that have the same name as the class. You can see a full list of your project files in the **Solution Explorer** window, visible on the side of the IDE. If the window isn't visible, you can open it from the menu bar: choose **View** > **Solution Explorer**.



You should now have three tabs open in the editor: *CalculatorTutorial.cpp*, *Calculator.h*, and *Calculator.cpp*. If you accidentally close one of them, you can reopen it by double-clicking it in the **Solution Explorer** window.

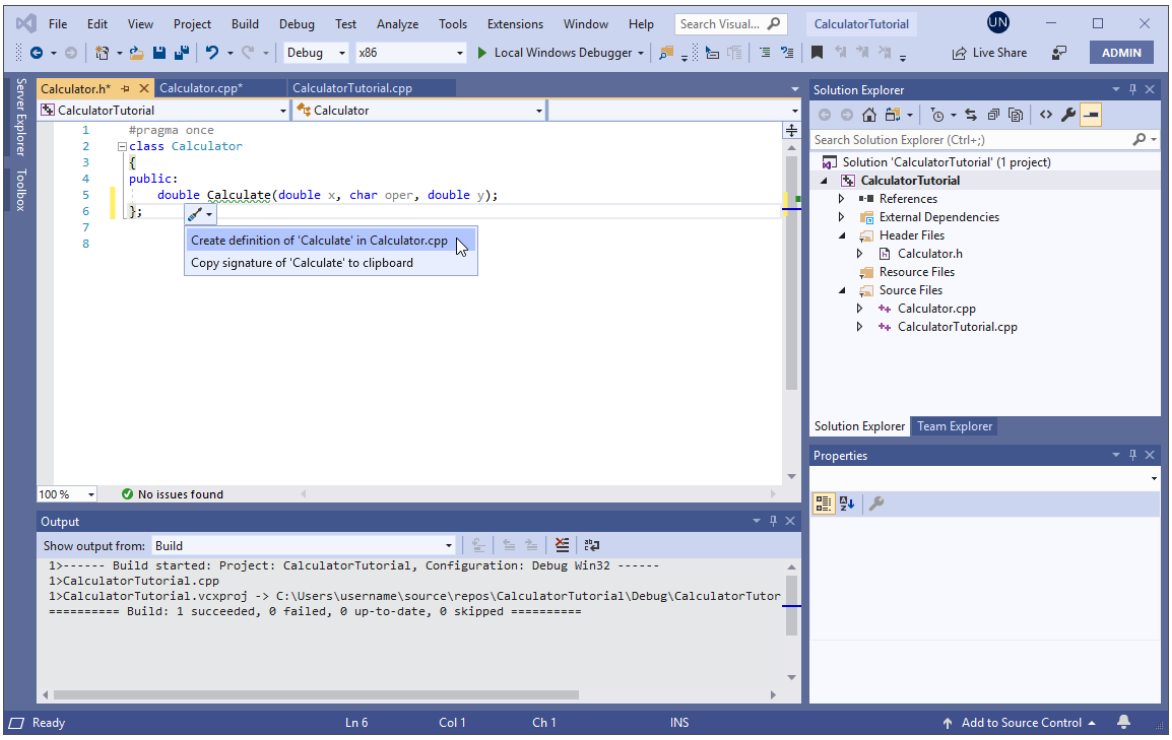
2. In **Calculator.h**, remove the `Calculator();` and `~Calculator();` lines that were generated, since you won't need them here. Next, add the following line of code so the file now looks like this:

```
#pragma once
class Calculator
{
public:
    double Calculate(double x, char oper, double y);
};
```

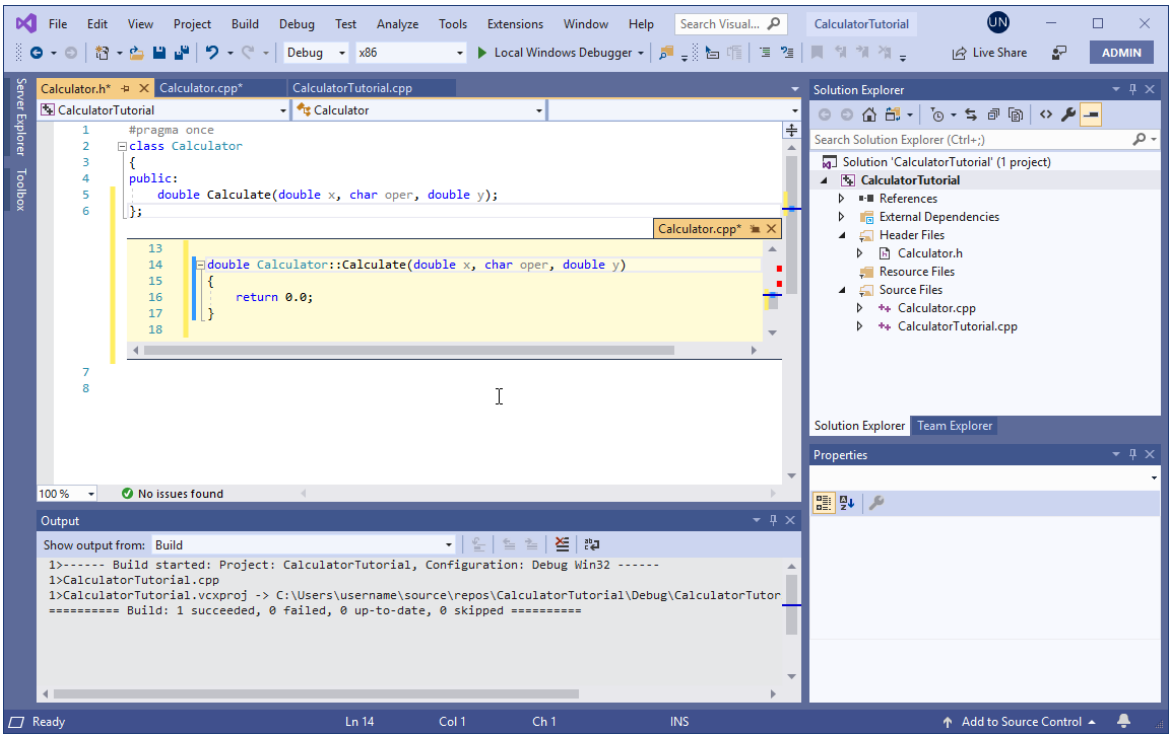
Understanding the code

- The line you added declares a new function called `Calculate`, which we'll use to run math operations for addition, subtraction, multiplication, and division.
- C++ code is organized into *header* (.h) files and *source* (.cpp) files. Several other file extensions are supported by various compilers, but these are the main ones to know about. Functions and variables are normally *declared*, that is, given a name and a type, in header files, and *implemented*, or given a definition, in source files. To access code defined in another file, you can use `#include "filename.h"`, where 'filename.h' is the name of the file that declares the variables or functions you want to use.
- The two lines you deleted declared a *constructor* and *destructor* for the class. For a simple class like this one, the compiler creates them for you, and their uses are beyond the scope of this tutorial.
- It's good practice to organize your code into different files based on what it does, so it's easy to find the code you need later. In our case, we define the `Calculator` class separately from the file containing the `main()` function, but we plan to reference the `Calculator` class in `main()`.

3. You'll see a green squiggle appear under `Calculate`. It's because we haven't defined the `Calculate` function in the .cpp file. Hover over the word, click the lightbulb (in this case, a screwdriver) that pops up, and choose **Create definition of 'Calculate' in Calculator.cpp**.



A pop-up appears that gives you a peek of the code change that was made in the other file. The code was added to *Calculator.cpp*.



Currently, it just returns 0.0. Let's change that. Press **Esc** to close the pop-up.

4. Switch to the *Calculator.cpp* file in the editor window. Remove the `Calculator()` and `~Calculator()` sections (as you did in the .h file) and add the following code to `Calculate()`:

```
#include "Calculator.h"

double Calculator::Calculate(double x, char oper, double y)
{
    switch(oper)
    {
        case '+':
            return x + y;
        case '-':
            return x - y;
        case '*':
            return x * y;
        case '/':
            return x / y;
        default:
            return 0.0;
    }
}
```

Understanding the code

- The function `Calculate` consumes a number, an operator, and a second number, then performs the requested operation on the numbers.
- The switch statement checks which operator was provided, and only executes the case corresponding to that operation. The default: case is a fallback in case the user types an operator that isn't accepted, so the program doesn't break. In general, it's best to handle invalid user input in a more elegant way, but this is beyond the scope of this tutorial.
- The `double` keyword denotes a type of number that supports decimals. This way, the calculator can handle both decimal math and integer math. The `Calculate` function is required to always return such a number due to the `double` at the very start of the code (this denotes the function's return type), which is why we return 0.0 even in the default case.
- The .h file declares the function *prototype*, which tells the compiler upfront what parameters it

requires, and what return type to expect from it. The .cpp file has all the implementation details of the function.

If you build and run the code again at this point, it will still exit after asking which operation to perform. Next, you'll modify the `main` function to do some calculations.

To call the Calculator class member functions

1. Now let's update the `main` function in *CalculatorTutorial.cpp*:

```
// CalculatorTutorial.cpp : This file contains the 'main' function. Program execution begins and ends
there.
//

#include <iostream>
#include "Calculator.h"

using namespace std;

int main()
{
    double x = 0.0;
    double y = 0.0;
    double result = 0.0;
    char oper = '+';

    cout << "Calculator Console Application" << endl << endl;
    cout << "Please enter the operation to perform. Format: a+b | a-b | a*b | a/b"
         << endl;

    Calculator c;
    while (true)
    {
        cin >> x >> oper >> y;
        result = c.Calculate(x, oper, y);
        cout << "Result is: " << result << endl;
    }

    return 0;
}
```

Understanding the code

- Since C++ programs always start at the `main()` function, we need to call our other code from there, so a `#include` statement is needed.
- Some initial variables `x`, `y`, `oper`, and `result` are declared to store the first number, second number, operator, and final result, respectively. It is always good practice to give them some initial values to avoid undefined behavior, which is what is done here.
- The `Calculator c;` line declares an object named 'c' as an instance of the `Calculator` class. The class itself is just a blueprint for how calculators work; the object is the specific calculator that does the math.
- The `while (true)` statement is a loop. The code inside the loop continues to execute over and over again as long as the condition inside the `()` holds true. Since the condition is simply listed as `true`, it's always true, so the loop runs forever. To close the program, the user must manually close the console window. Otherwise, the program always waits for new input.
- The `cin` keyword is used to accept input from the user. This input stream is smart enough to process a line of text entered in the console window and place it inside each of the variables listed, in order, assuming the user input matches the required specification. You can modify this line to accept different types of input, for instance, more than two numbers, though the `Calculate()`

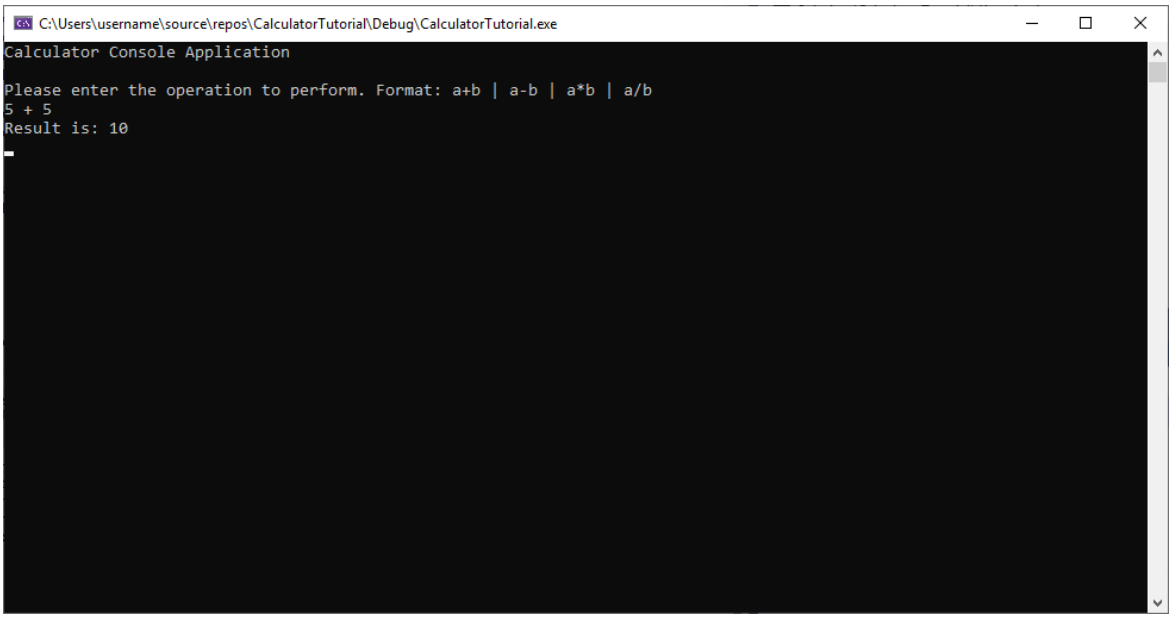
function would also need to be updated to handle this.

- The `c.Calculate(x, oper, y);` expression calls the `Calculate` function defined earlier, and supplies the entered input values. The function then returns a number that gets stored in `result`.
- Finally, `result` is printed to the console, so the user sees the result of the calculation.

Build and test the code again

Now it's time to test the program again to make sure everything works properly.

1. Press **Ctrl+F5** to rebuild and start the app.
2. Enter `5 + 5`, and press **Enter**. Verify that the result is 10.

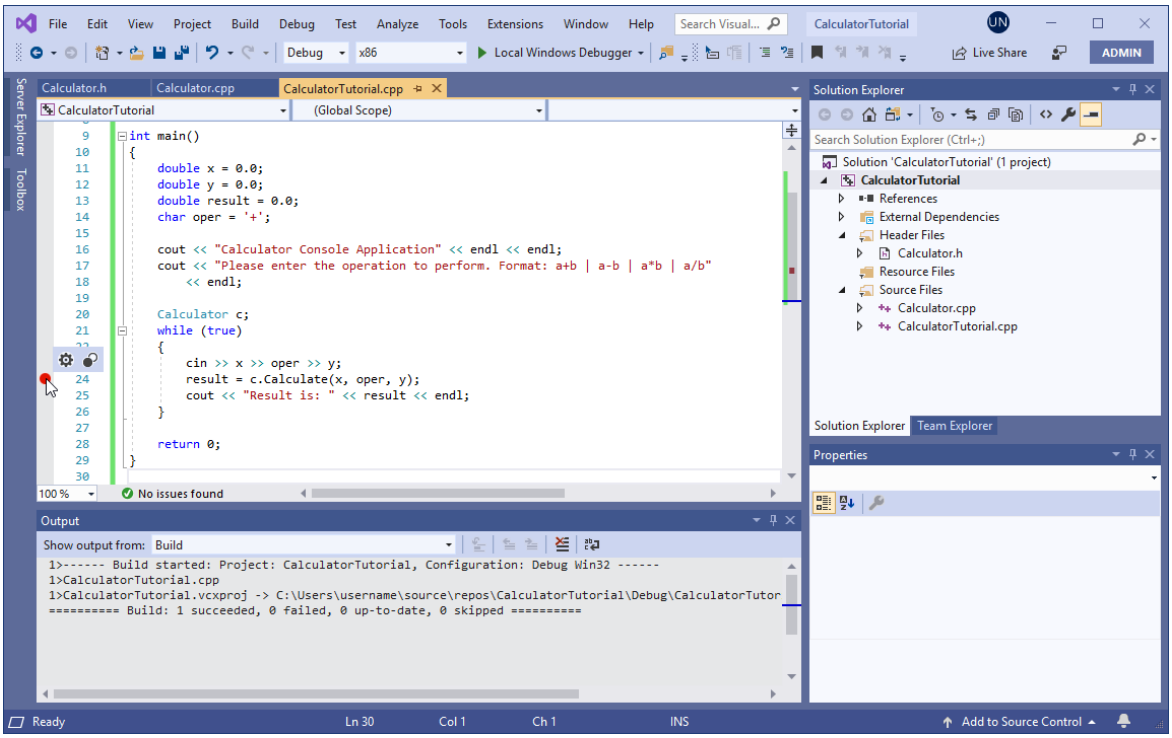


Debug the app

Since the user is free to type anything into the console window, let's make sure the calculator handles some input as expected. Instead of running the program, let's debug it instead, so we can inspect what it's doing in detail, step-by-step.

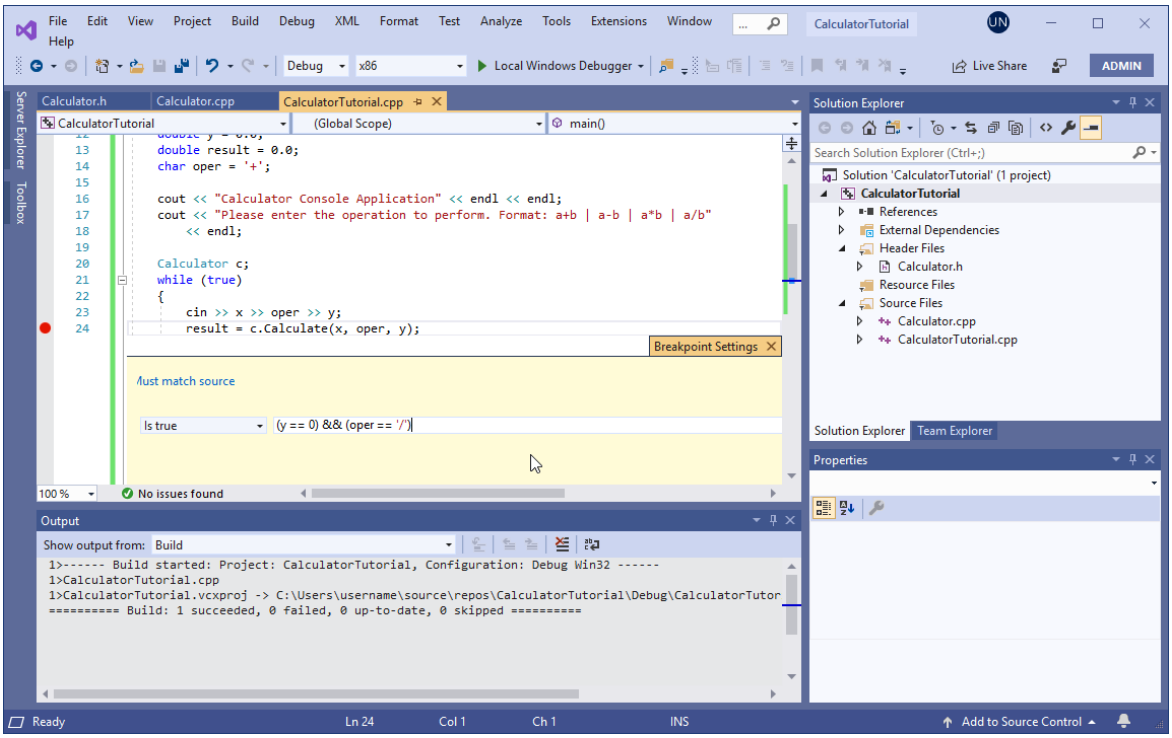
To run the app in the debugger

1. Set a breakpoint on the `result = c.Calculate(x, oper, y);` line, just after the user was asked for input. To set the breakpoint, click next to the line in the gray vertical bar along the left edge of the editor window. A red dot appears.



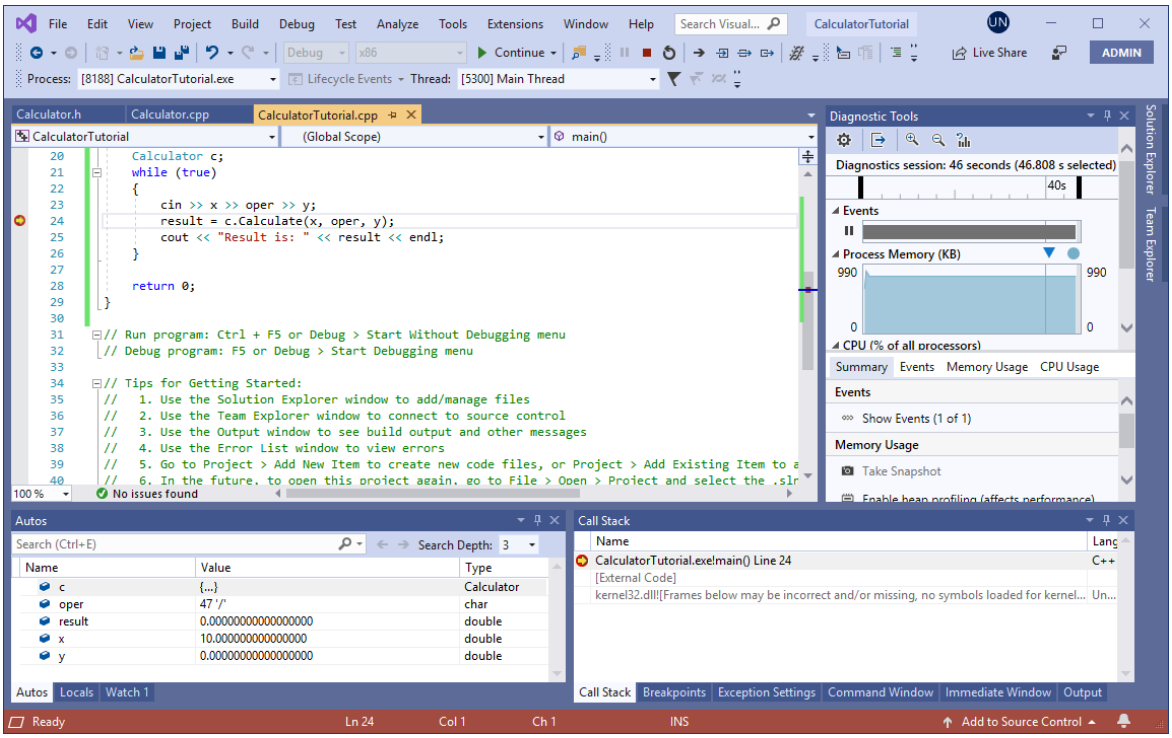
Now when we debug the program, it always pauses execution at that line. We already have a rough idea that the program works for simple cases. Since we don't want to pause execution every time, let's make the breakpoint conditional.

2. Right-click the red dot that represents the breakpoint, and choose **Conditions**. In the edit box for the condition, enter `(y == 0) && (oper == '/')`. Choose the **Close** button when you're done. The condition is saved automatically.



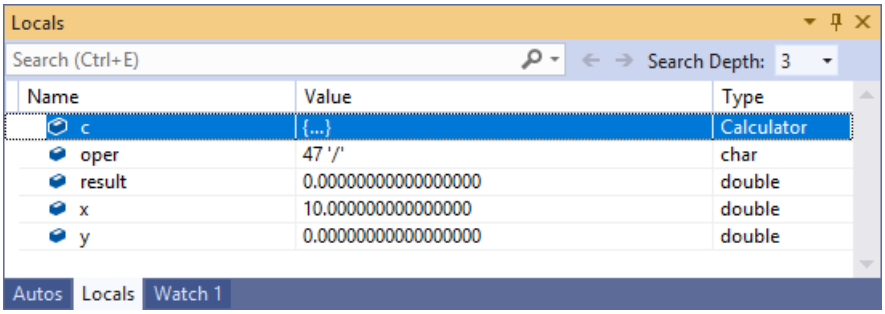
Now we pause execution at the breakpoint specifically if a division by 0 is attempted.

3. To debug the program, press **F5**, or choose the **Local Windows Debugger** toolbar button that has the green arrow icon. In your console app, if you enter something like "5 - 0", the program behaves normally and keeps running. However, if you type "10 / 0", it pauses at the breakpoint. You can even put any number of spaces between the operator and numbers: `cin` is smart enough to parse the input appropriately.

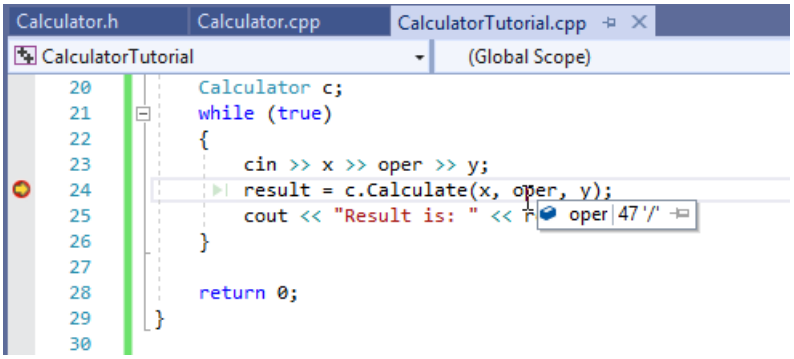


Useful windows in the debugger

Whenever you debug your code, you may notice that some new windows appear. These windows can assist your debugging experience. Take a look at the **Autos** window. The **Autos** window shows you the current values of variables used at least three lines before and up to the current line. To see all of the variables from that function, switch to the **Locals** window. You can actually modify the values of these variables while debugging, to see what effect they would have on the program. In this case, we'll leave them alone.



You can also just hover over variables in the code itself to see their current values where the execution is currently paused. Make sure the editor window is in focus by clicking on it first.



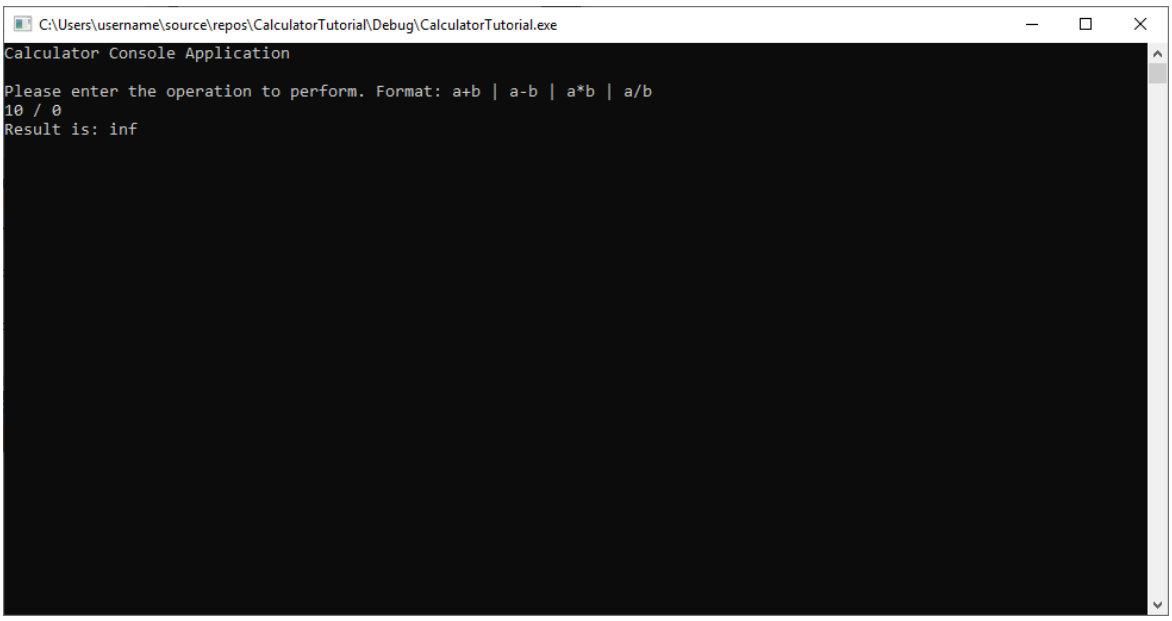
To continue debugging

1. The yellow line on the left shows the current point of execution. The current line calls `Calculate`, so press **F11** to **Step Into** the function. You'll find yourself in the body of the `Calculate` function. Be careful with **Step Into**; if you do it too much, you may waste a lot of time. It goes into any code you use on the line you are on, including standard library functions.

2. Now that the point of execution is at the start of the `Calculate` function, press **F10** to move to the next line in the program's execution. **F10** is also known as **Step Over**. You can use **Step Over** to move from line to line, without delving into the details of what is occurring in each part of the line. In general you should use **Step Over** instead of **Step Into**, unless you want to dive more deeply into code that is being called from elsewhere (as you did to reach the body of `Calculate`).

3. Continue using **F10** to **Step Over** each line until you get back to the `main()` function in the other file, and stop on the `cout` line.

   It looks like the program is doing what is expected: it takes the first number, and divides it by the second. On the `cout` line, hover over the `result` variable or take a look at `result` in the **Autos** window. You'll see its value is listed as "inf", which doesn't look right, so let's fix it. The `cout` line just outputs whatever value is stored in `result`, so when you step one more line forward using **F10**, the console window displays:



   This result happens because division by zero is undefined, so the program doesn't have a numerical answer to the requested operation.

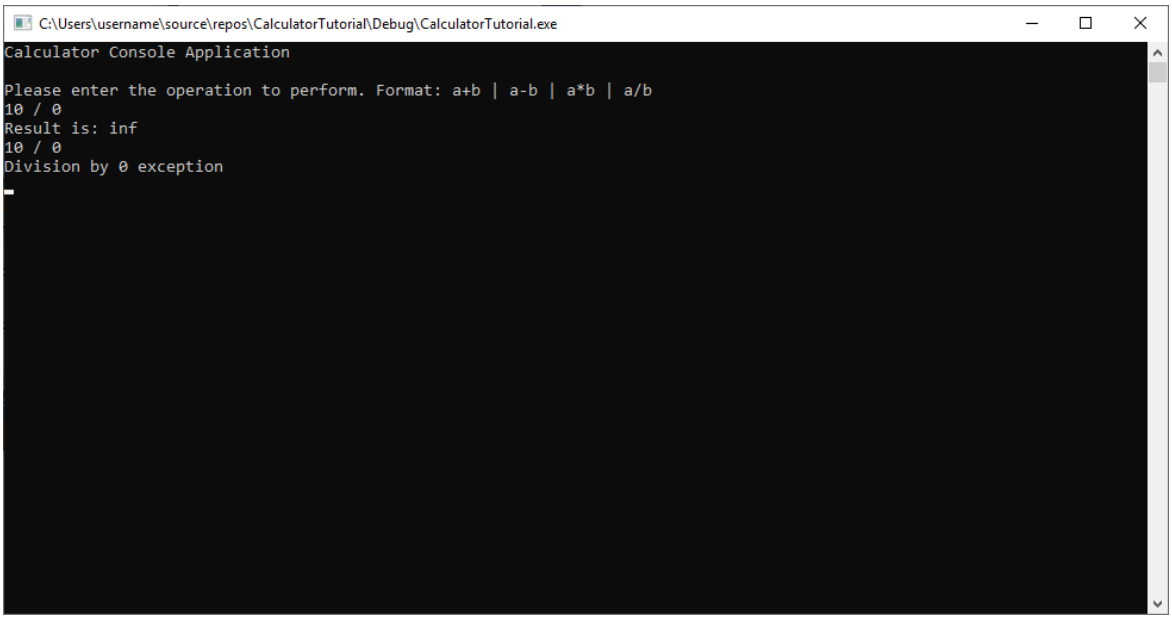
To fix the "divide by zero" error

Let's handle division by zero more gracefully, so a user can understand the problem.

1. Make the following changes in *CalculatorTutorial.cpp*. (You can leave the program running as you edit, thanks to a debugger feature called **Edit and Continue**):

```
// CalculatorTutorial.cpp : This file contains the 'main' function. Program execution begins and ends
there.
//

#include <iostream>
#include "Calculator.h"

using namespace std;

int main()
{
    double x = 0.0;
    double y = 0.0;
    double result = 0.0;
    char oper = '+';

    cout << "Calculator Console Application" << endl << endl;
    cout << "Please enter the operation to perform. Format: a+b | a-b | a*b | a/b" << endl;

    Calculator c;
    while (true)
    {
        cin  >> x  >> oper  >> y;
        if (oper == '/' && y == 0)
        {
            cout << "Division by 0 exception" << endl;
            continue;
        }
        else
        {
            result = c.Calculate(x, oper, y);
        }
        cout << "Result is: " << result << endl;
    }

    return 0;
}
```

2. Now press **F5** once. Program execution continues all the way until it has to pause to ask for user input. Enter `10 / 0` again. Now, a more helpful message is printed. The user is asked for more input, and the program continues executing normally.



NOTE

When you edit code while in debugging mode, there is a risk of code becoming stale. This happens when the debugger is still running your old code, and has not yet updated it with your changes. The debugger pops up a dialog to inform you when this happens. Sometimes, you may need to press **F5** to refresh the code being executed. In particular, if you make a change inside a function while the point of execution is inside that function, you'll need to step out of the function, then back into it again to get the updated code. If that doesn't work for some reason and you see an error message, you can stop debugging by clicking on the red square in the toolbar under the menus at the top of the IDE, then start debugging again by entering **F5** or by choosing the green "play" arrow beside the stop button on the toolbar.

Understanding the Run and Debug shortcuts

- **F5** (or **Debug** > **Start Debugging**) starts a debugging session if one isn't already active, and runs the program until a breakpoint is hit or the program needs user input. If no user input is needed and no breakpoint is available to hit, the program terminates and the console window closes itself when the program finishes running. If you have something like a "Hello World" program to run, use **Ctrl+F5** or set a breakpoint before you enter **F5** to keep the window open.
- **Ctrl+F5** (or **Debug** > **Start Without Debugging**) runs the application without going into debug mode. This is slightly faster than debugging, and the console window stays open after the program finishes executing.
- **F10** (known as **Step Over**) lets you iterate through code, line-by-line, and visualize how the code is run and what variable values are at each step of execution.
- **F11** (known as **Step Into**) works similarly to **Step Over**, except it steps into any functions called on the line of execution. For example, if the line being executed calls a function, pressing **F11** moves the pointer into the body of the function, so you can follow the function's code being run before coming back to the line you started at. Pressing **F10** steps over the function call and just moves to the next line; the function call still happens, but the program doesn't pause to show you what it's doing.

Close the app

- If it's still running, close the console window for the calculator app.

Add Git source control

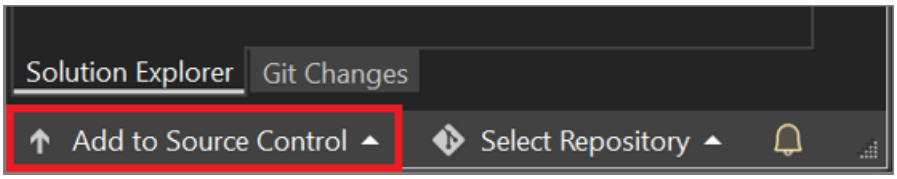
Now that you've created an app, you might want to add it to a Git repository. We've got you covered. Visual Studio makes that process easy with Git tools you can use directly from the IDE.

TIP

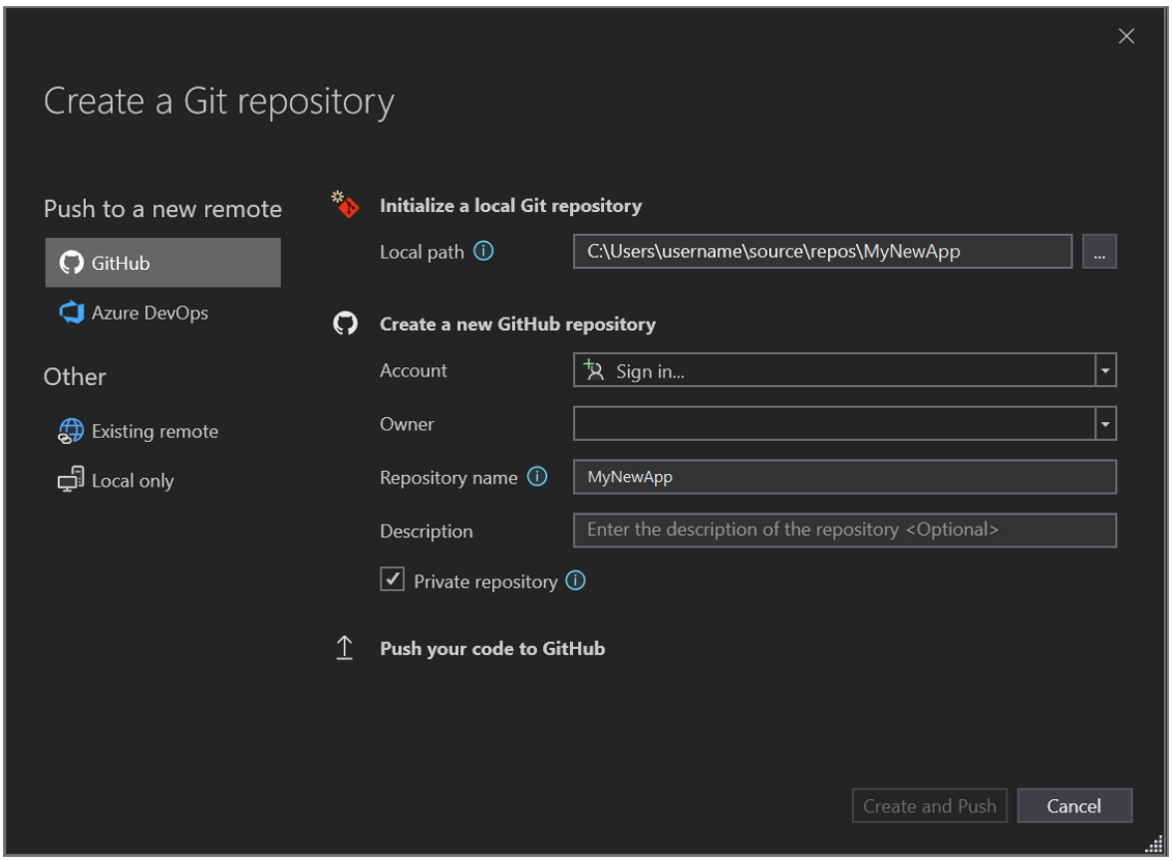
Git is the most widely used modern version control system, so whether you're a professional developer or you're learning how to code, Git can be very useful. If you're new to Git, the https://git-scm.com/ website is a good place to start. There, you can find cheat sheets, a popular online book, and Git Basics videos.

To associate your code with Git, you start by creating a new Git repository where your code is located. Here's how:

1. In the status bar at the bottom-right corner of Visual Studio, select **Add to Source Control**, and then select **Git**.



2. In the **Create a Git repository** dialog box, sign in to GitHub.

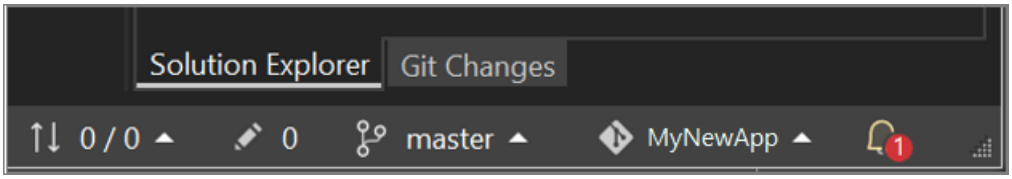


The repository name auto-populates based on your folder location. By default, your new repository is private, which means you're the only one who can access it.

> **TIP**
> Whether your repository is public or private, it's best to have a remote backup of your code stored securely on GitHub. Even if you aren't working with a team, a remote repository makes your code available to you from any computer.

3. Select **Create and Push**.

After you create your repository, you see status details in the status bar.



The first icon with the arrows shows how many outgoing/incoming commits are in your current branch. You can use this icon to pull any incoming commits or push any outgoing commits. You can also choose to view these commits first. To do so, select the icon, and then select **View Outgoing/Incoming**.

The second icon with the pencil shows the number of uncommitted changes to your code. You can select this icon to view those changes in the **Git Changes** window.

To learn more about how to use Git with your app, see the Visual Studio version control documentation.

The finished app

Congratulations! You've completed the code for the calculator app, built and debugged it, and added it to a repo, all in Visual Studio.

Next steps

Learn more about Visual Studio for C++

The usual starting point for a C++ programmer is a "Hello, world!" application that runs on the command line. That's what you'll create in Visual Studio in this article, and then we'll move on to something more challenging: a calculator app.

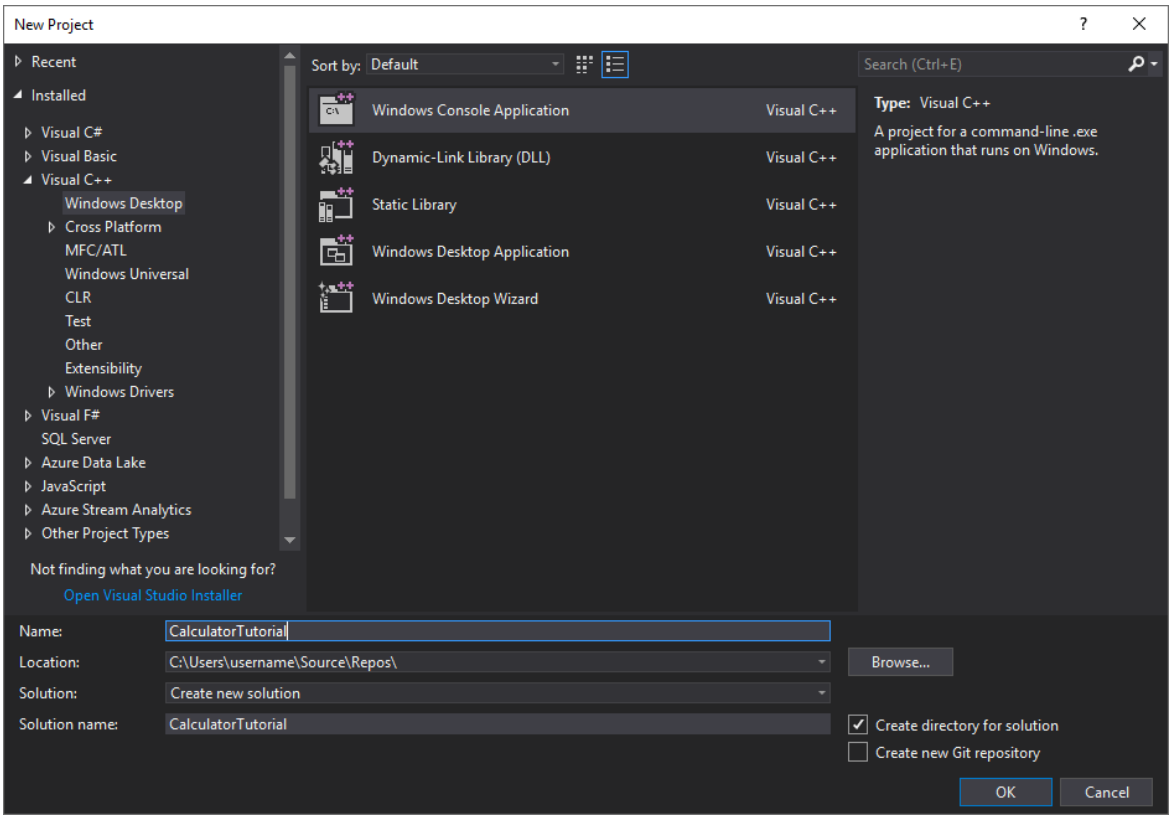
Prerequisites

- Have Visual Studio with the **Desktop development with C++** workload installed and running on your computer. If it's not installed yet, see Install C++ support in Visual Studio.

Create your app project

Visual Studio uses *projects* to organize the code for an app, and *solutions* to organize your projects. A project contains all the options, configurations, and rules used to build your apps. It also manages the relationship between all the project's files and any external files. To create your app, first, you'll create a new project and solution.

1. On the menubar in Visual Studio, choose **File** > **New** > **Project**. The **New Project** window opens.
2. On the left sidebar, make sure **Visual C++** is selected. In the center, choose **Windows Console Application**.
3. In the **Name** edit box at the bottom, name the new project *CalculatorTutorial*, then choose **OK**.



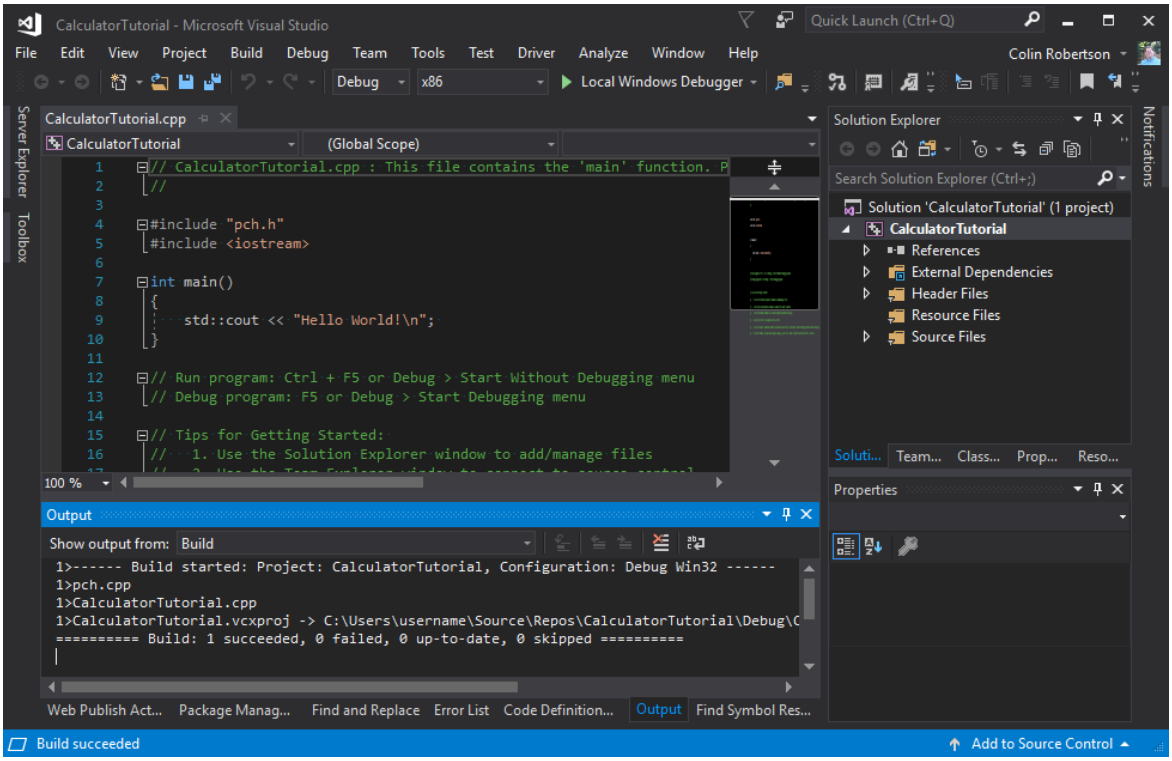
An empty C++ Windows console application gets created. Console applications use a Windows console window to display output and accept user input. In Visual Studio, an editor window opens and shows the generated code:

```
// CalculatorTutorial.cpp : This file contains the 'main' function. Program execution begins and ends
there.
//

#include "pch.h"
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu

// Tips for Getting Started:
//   1. Use the Solution Explorer window to add/manage files
//   2. Use the Team Explorer window to connect to source control
//   3. Use the Output window to see build output and other messages
//   4. Use the Error List window to view errors
//   5. Go to Project > Add New Item to create new code files, or Project > Add Existing Item to add
existing code files to the project
//   6. In the future, to open this project again, go to File > Open > Project and select the .sln
file
```

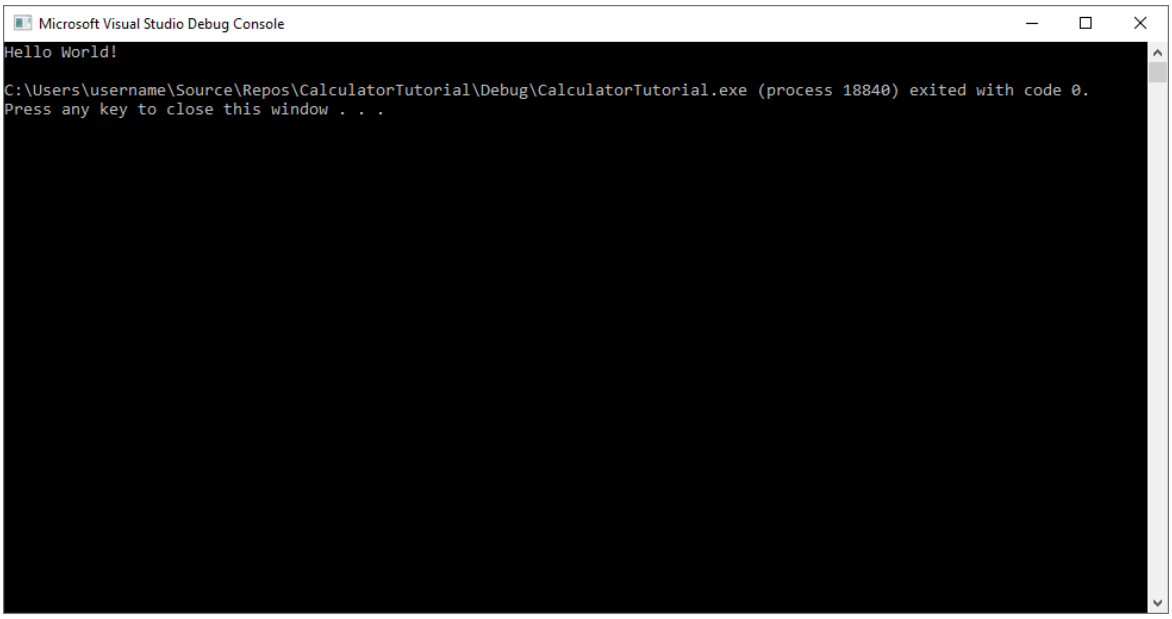
Verify that your new app builds and runs

The template for a new windows console application creates a simple C++ "Hello World" app. At this point, you can see how Visual Studio builds and runs the apps you create right from the IDE.

1. To build your project, choose **Build Solution** from the **Build** menu. The **Output** window shows the results of the build process.



2. To run the code, on the menu bar, choose **Debug**, **Start without debugging**.



A console window opens and then runs your app. When you start a console app in Visual Studio, it runs your code, then prints "Press any key to continue . . ." to give you a chance to see the output. Congratulations! You've created your first "Hello, world!" console app in Visual Studio!

3. Press a key to dismiss the console window and return to Visual Studio.

You now have the tools to build and run your app after every change, to verify that the code still works as you expect. Later, we'll show you how to debug it if it doesn't.

Edit the code

Now let's turn the code in this template into a calculator app.

1. In the *CalculatorTutorial.cpp* file, edit the code to match this example:

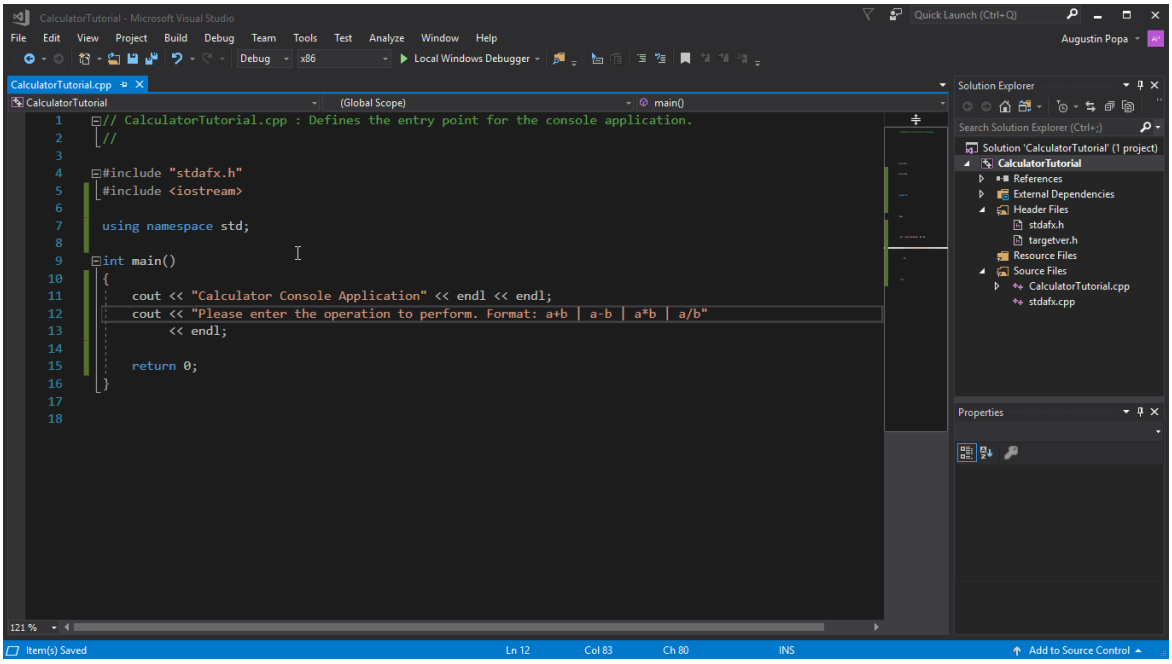
```
// CalculatorTutorial.cpp : This file contains the 'main' function. Program execution begins and ends
there.
//

#include "pch.h"
#include <iostream>

using namespace std;

int main()
{
    cout << "Calculator Console Application" << endl << endl;
    cout << "Please enter the operation to perform. Format: a+b | a-b | a*b | a/b"
        << endl;
    return 0;
}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu
// Tips for Getting Started:
//   1. Use the Solution Explorer window to add/manage files
//   2. Use the Team Explorer window to connect to source control
//   3. Use the Output window to see build output and other messages
//   4. Use the Error List window to view errors
//   5. Go to Project > Add New Item to create new code files, or Project > Add Existing Item to add
existing code files to the project
//   6. In the future, to open this project again, go to File > Open > Project and select the .sln
file
```

Understanding the code:

- The `#include` statements allow you to reference code located in other files. Sometimes, you may see a filename surrounded by angle brackets (< >); other times, it's surrounded by quotes (" "). In general, angle brackets are used when referencing the C++ Standard Library, while quotes are used for other files.
- The `#include "pch.h"` (or in Visual Studio 2017 and earlier, `#include "stdafx.h"`) line references something known as a precompiled header. These are often used by professional programmers to improve compilation times, but they are beyond the scope of this tutorial.
- The `using namespace std;` line tells the compiler to expect stuff from the C++ Standard Library to be used in this file. Without this line, each keyword from the library would have to be preceded with a `std::`, to denote its scope. For instance, without that line, each reference to `cout` would have to be written as `std::cout`. The `using` statement is added to make the code look more clean.
- The `cout` keyword is used to print to standard output in C++. The `<<` operator tells the compiler to send whatever is to the right of it to the standard output.
- The **endl** keyword is like the Enter key; it ends the line and moves the cursor to the next line. It is a better practice to put a `\n` inside the string (contained by "") to do the same thing, as `endl` always flushes the buffer and can hurt the performance of the program, but since this is a very small app, `endl` is used instead for better readability.
- All C++ statements must end with semicolons and all C++ applications must contain a `main()` function. This function is what the program runs at the start. All code must be accessible from `main()` in order to be used.

2. To save the file, enter **Ctrl+S**, or choose the **Save** icon near the top of the IDE, the floppy disk icon in the toolbar under the menu bar.

3. To run the application, press **Ctrl+F5** or go to the **Debug** menu and choose **Start Without Debugging**. If you get a pop-up that says **This project is out of date**, you may select **Do not show this dialog again**, and then choose **Yes** to build your application. You should see a console window appear that displays the text specified in the code.



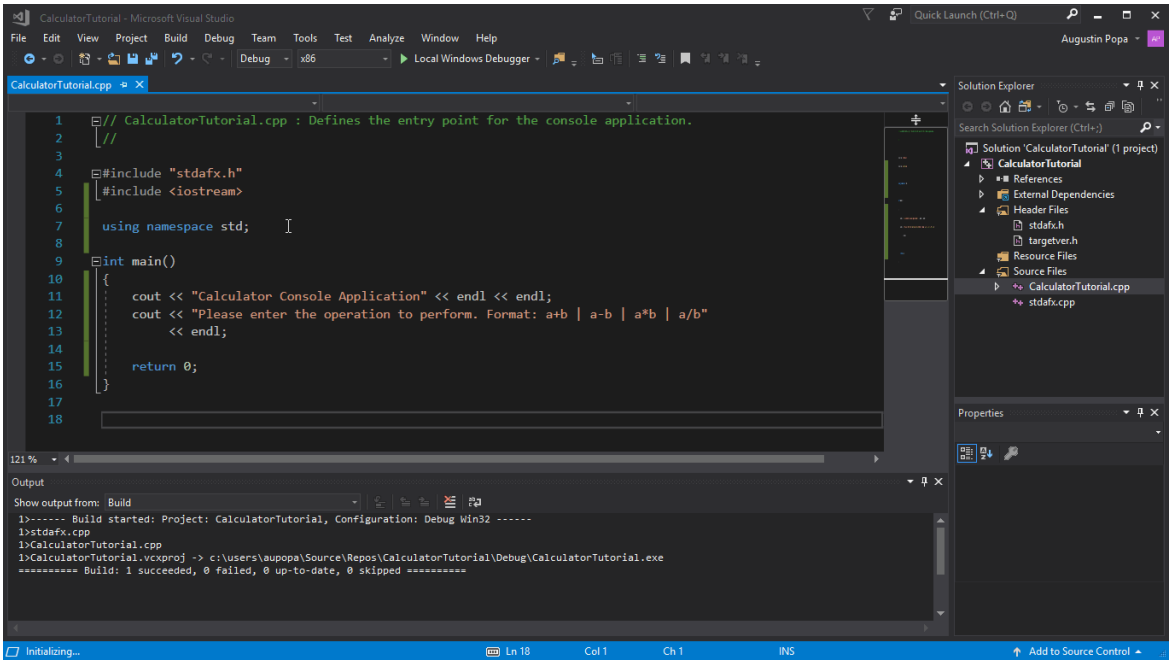
4. Close the console window when you're done.

Add code to do some math

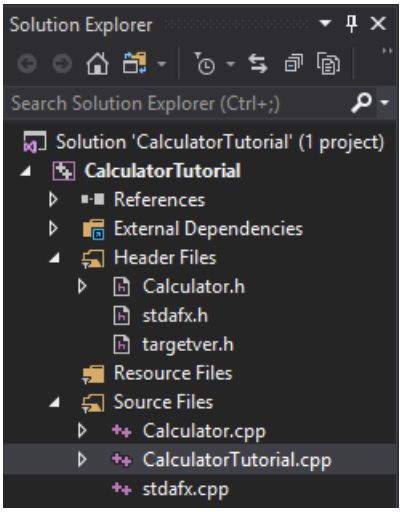
It's time to add some math logic.

To add a Calculator class

1. Go to the **Project** menu and choose **Add Class**. In the **Class Name** edit box, enter *Calculator*. Choose **OK**. Two new files get added to your project. To save all your changed files at once, press **Ctrl+Shift+S**. It's a keyboard shortcut for **File** > **Save All**. There's also a toolbar button for **Save All**, an icon of two floppy disks, found beside the **Save** button. In general, it's good practice to do **Save All** frequently, so you don't miss any files when you save.



A class is like a blueprint for an object that does something. In this case, we define a calculator and how it should work. The **Add Class** wizard you used above created .h and .cpp files that have the same name as the class. You can see a full list of your project files in the **Solution Explorer** window, visible on the side of the IDE. If the window isn't visible, you can open it from the menu bar: choose **View** > **Solution Explorer**.



You should now have three tabs open in the editor: *CalculatorTutorial.cpp*, *Calculator.h*, and *Calculator.cpp*. If you accidentally close one of them, you can reopen it by double-clicking it in the **Solution Explorer** window.

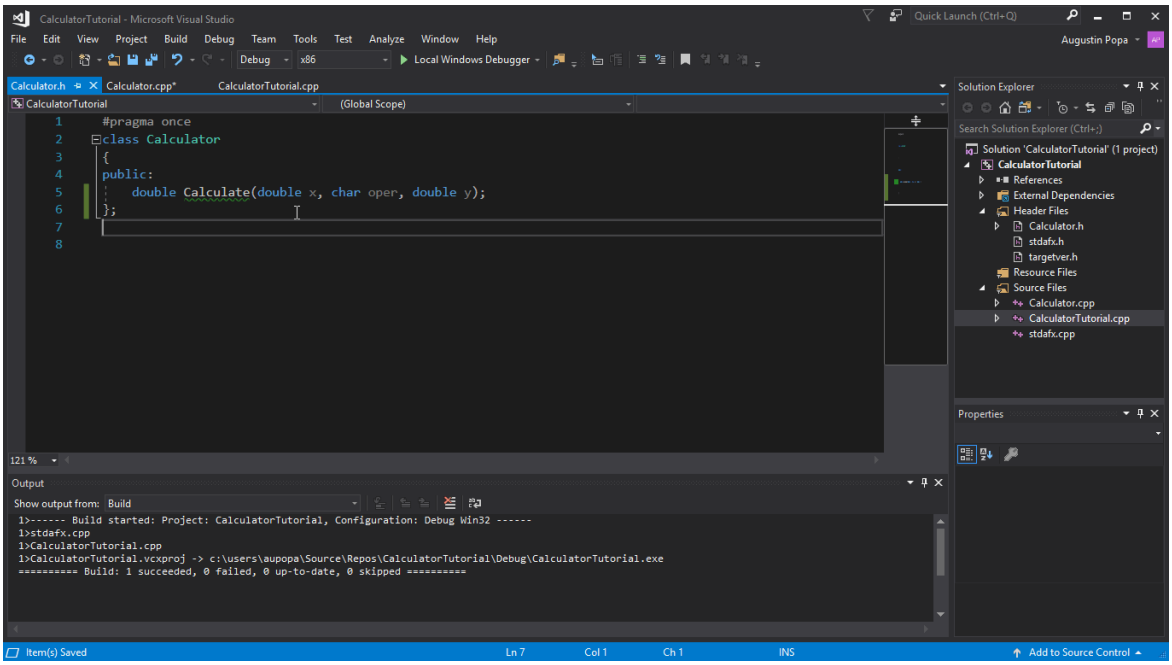
2. In **Calculator.h**, remove the `Calculator();` and `~Calculator();` lines that were generated, since you won't need them here. Next, add the following line of code so the file now looks like this:

```
#pragma once
class Calculator
{
public:
    double Calculate(double x, char oper, double y);
};
```

Understanding the code

- The line you added declares a new function called `Calculate`, which we'll use to run math operations for addition, subtraction, multiplication, and division.
- C++ code is organized into *header* (.h) files and *source* (.cpp) files. Several other file extensions are supported by various compilers, but these are the main ones to know about. Functions and variables are normally *declared*, that is, given a name and a type, in header files, and *implemented*, or given a definition, in source files. To access code defined in another file, you can use `#include "filename.h"`, where 'filename.h' is the name of the file that declares the variables or functions you want to use.
- The two lines you deleted declared a *constructor* and *destructor* for the class. For a simple class like this one, the compiler creates them for you, and their uses are beyond the scope of this tutorial.
- It's good practice to organize your code into different files based on what it does, so it's easy to find the code you need later. In our case, we define the `Calculator` class separately from the file containing the `main()` function, but we plan to reference the `Calculator` class in `main()`.

3. You'll see a green squiggle appear under `Calculate`. It's because we haven't defined the `Calculate` function in the .cpp file. Hover over the word, click the lightbulb that pops up, and choose **Create definition of 'Calculate' in Calculator.cpp**. A pop-up appears that gives you a peek of the code change that was made in the other file. The code was added to *Calculator.cpp*.



Currently, it just returns 0.0. Let's change that. Press **Esc** to close the pop-up.

4. Switch to the *Calculator.cpp* file in the editor window. Remove the `Calculator()` and `~Calculator()` sections (as you did in the .h file) and add the following code to `Calculate()`:

```
#include "pch.h"
#include "Calculator.h"

double Calculator::Calculate(double x, char oper, double y)
{
    switch(oper)
    {
        case '+':
            return x + y;
        case '-':
            return x - y;
        case '*':
            return x * y;
        case '/':
            return x / y;
        default:
            return 0.0;
    }
}
```

Understanding the code

- The function `Calculate` consumes a number, an operator, and a second number, then performs the requested operation on the numbers.
- The switch statement checks which operator was provided, and only executes the case corresponding to that operation. The default: case is a fallback in case the user types an operator that isn't accepted, so the program doesn't break. In general, it's best to handle invalid user input in a more elegant way, but this is beyond the scope of this tutorial.
- The `double` keyword denotes a type of number that supports decimals. This way, the calculator can handle both decimal math and integer math. The `Calculate` function is required to always return such a number due to the `double` at the very start of the code (this denotes the function's return type), which is why we return 0.0 even in the default case.
- The .h file declares the function *prototype*, which tells the compiler upfront what parameters it requires, and what return type to expect from it. The .cpp file has all the implementation details of the function.

If you build and run the code again at this point, it will still exit after asking which operation to perform. Next, you'll modify the `main` function to do some calculations.

To call the Calculator class member functions

1. Now let's update the `main` function in *CalculatorTutorial.cpp*:

```
// CalculatorTutorial.cpp : This file contains the 'main' function. Program execution begins and ends
there.
//

#include "pch.h"
#include <iostream>
#include "Calculator.h"

using namespace std;

int main()
{
    double x = 0.0;
    double y = 0.0;
    double result = 0.0;
    char oper = '+';

    cout << "Calculator Console Application" << endl << endl;
    cout << "Please enter the operation to perform. Format: a+b | a-b | a*b | a/b"
         << endl;

    Calculator c;
    while (true)
    {
        cin >> x >> oper >> y;
        result = c.Calculate(x, oper, y);
        cout << "Result is: " << result << endl;
    }

    return 0;
}
```

Understanding the code

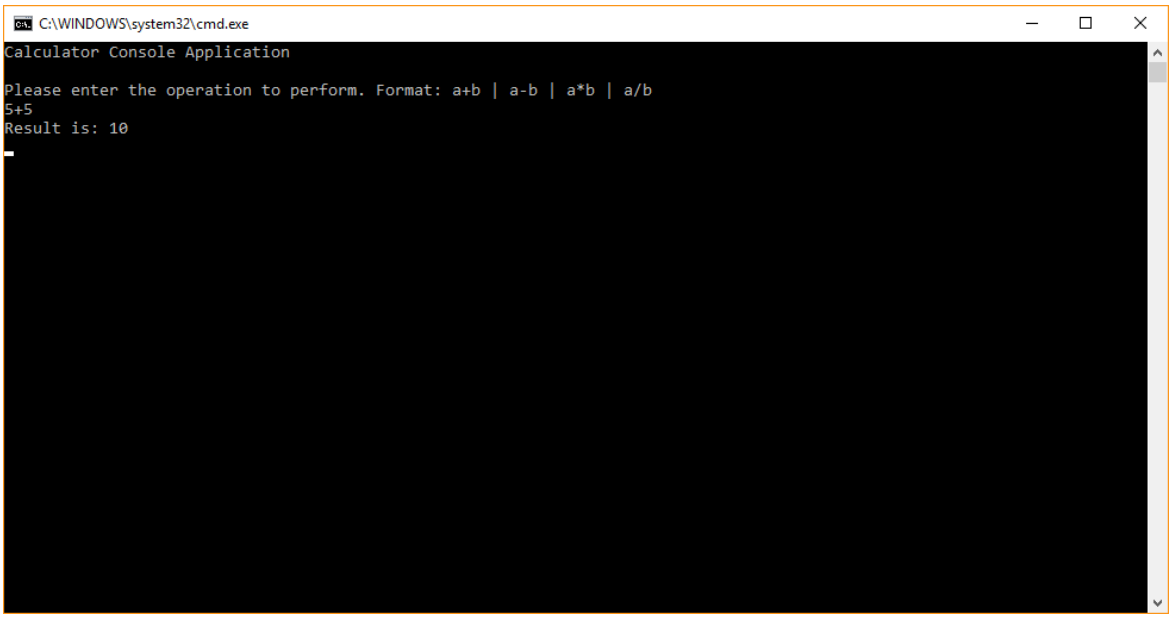
- Since C++ programs always start at the `main()` function, we need to call our other code from there, so a `#include` statement is needed.
- Some initial variables `x`, `y`, `oper`, and `result` are declared to store the first number, second number, operator, and final result, respectively. It is always good practice to give them some initial values to avoid undefined behavior, which is what is done here.
- The `Calculator c;` line declares an object named 'c' as an instance of the `Calculator` class. The class itself is just a blueprint for how calculators work; the object is the specific calculator that does the math.
- The `while (true)` statement is a loop. The code inside the loop continues to execute over and over again as long as the condition inside the `()` holds true. Since the condition is simply listed as `true`, it's always true, so the loop runs forever. To close the program, the user must manually close the console window. Otherwise, the program always waits for new input.
- The `cin` keyword is used to accept input from the user. This input stream is smart enough to process a line of text entered in the console window and place it inside each of the variables listed, in order, assuming the user input matches the required specification. You can modify this line to accept different types of input, for instance, more than two numbers, though the `Calculate()` function would also need to be updated to handle this.
- The `c.Calculate(x, oper, y);` expression calls the `Calculate` function defined earlier, and supplies the entered input values. The function then returns a number that gets stored in `result`.
- Finally, `result` is printed to the console, so the user sees the result of the calculation.

Build and test the code again

Now it's time to test the program again to make sure everything works properly.

1. Press **Ctrl+F5** to rebuild and start the app.

2. Enter `5 + 5`, and press **Enter**. Verify that the result is 10.

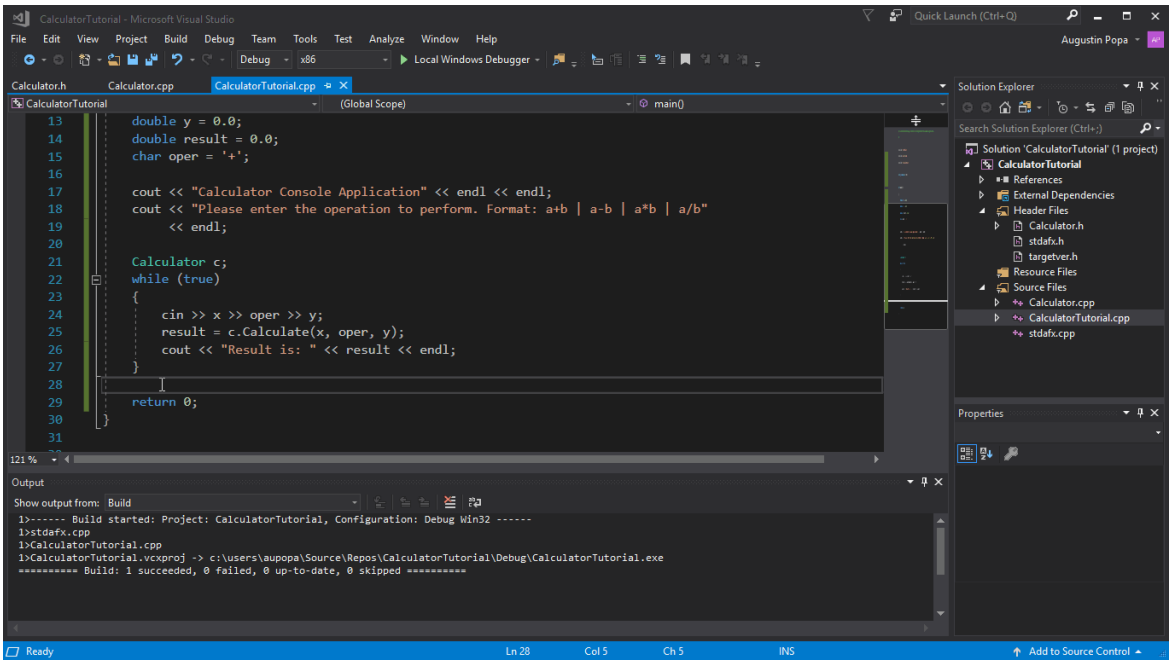


Debug the app

Since the user is free to type anything into the console window, let's make sure the calculator handles some input as expected. Instead of running the program, let's debug it instead, so we can inspect what it's doing in detail, step-by-step.

To run the app in the debugger

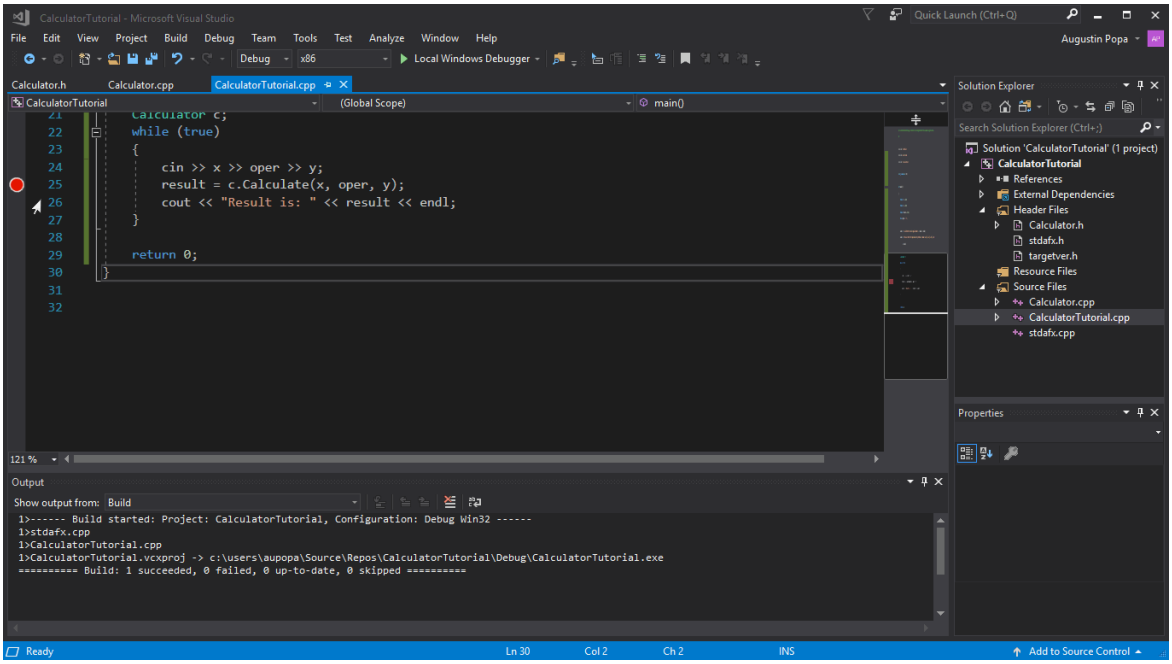
1. Set a breakpoint on the `result = c.Calculate(x, oper, y);` line, just after the user was asked for input. To set the breakpoint, click next to the line in the gray vertical bar along the left edge of the editor window. A red dot appears.



   Now when we debug the program, it always pauses execution at that line. We already have a rough idea that the program works for simple cases. Since we don't want to pause execution every time, let's make the breakpoint conditional.

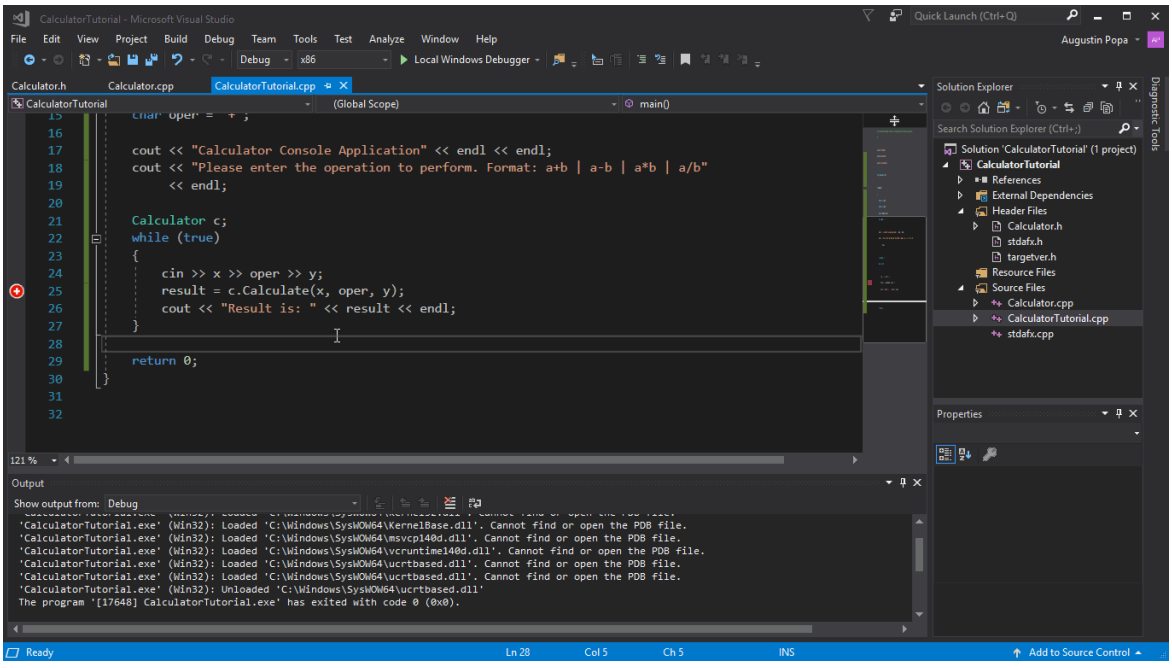
2. Right-click the red dot that represents the breakpoint, and choose **Conditions**. In the edit box for the

condition, enter `(y == 0) && (oper == '/')`. Choose the **Close** button when you're done. The condition is saved automatically.



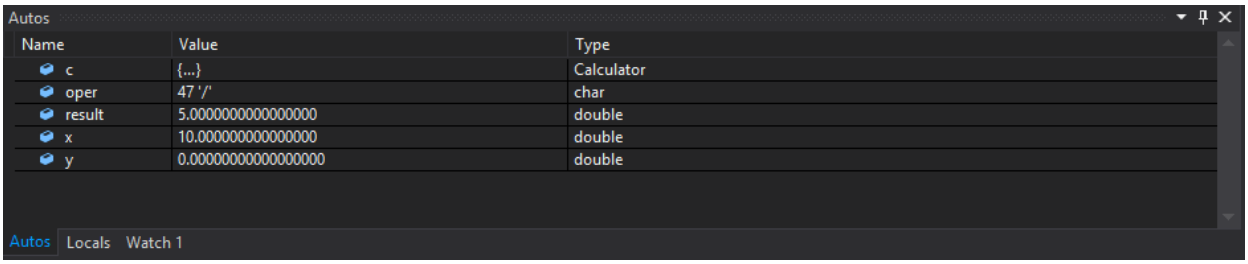
Now we pause execution at the breakpoint specifically if a division by 0 is attempted.

3. To debug the program, press **F5**, or choose the **Local Windows Debugger** toolbar button that has the green arrow icon. In your console app, if you enter something like "5 - 0", the program behaves normally and keeps running. However, if you type "10 / 0", it pauses at the breakpoint. You can even put any number of spaces between the operator and numbers; `cin` is smart enough to parse the input appropriately.

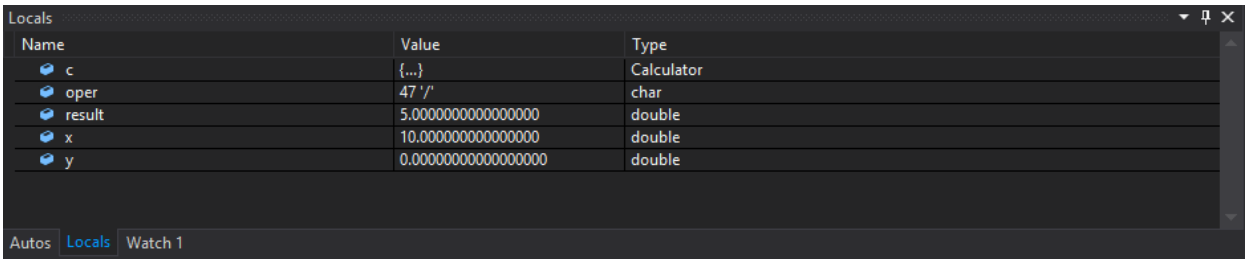


Useful windows in the debugger

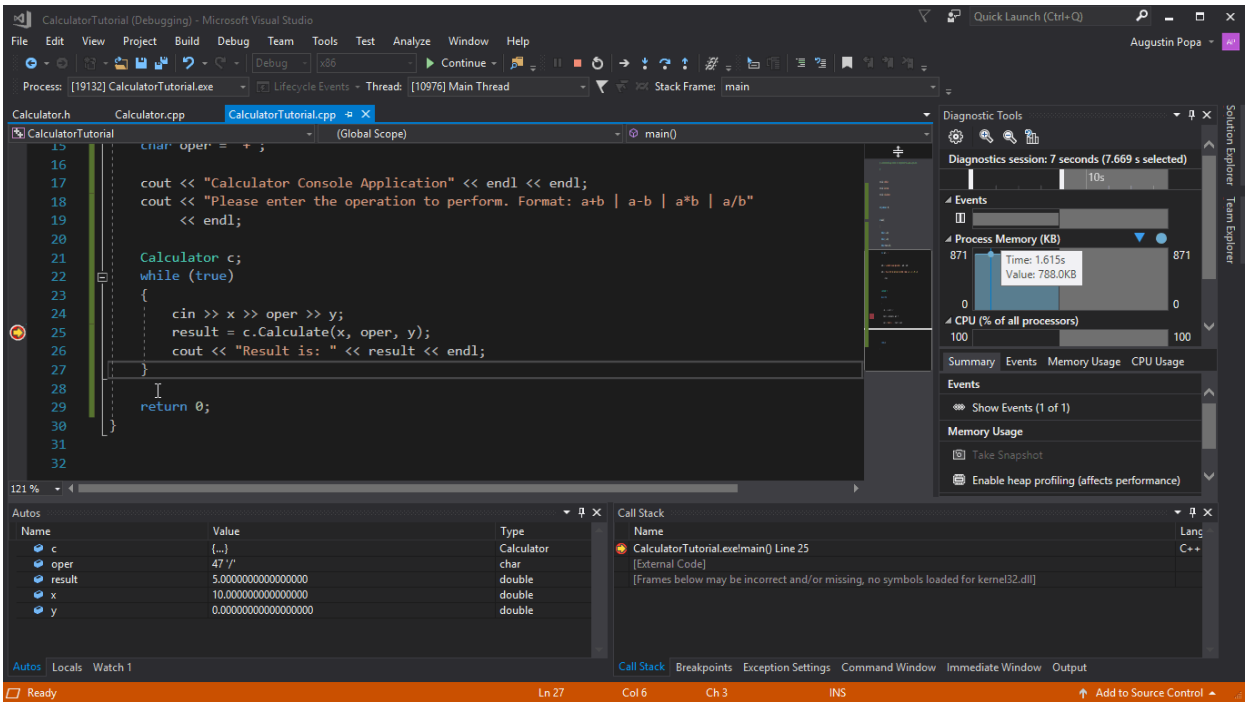
Whenever you debug your code, you may notice that some new windows appear. These windows can assist your debugging experience. Take a look at the **Autos** window. The **Autos** window shows you the current values of variables used at least three lines before and up to the current line.



To see all of the variables from that function, switch to the **Locals** window. You can actually modify the values of these variables while debugging, to see what effect they would have on the program. In this case, we'll leave them alone.



You can also just hover over variables in the code itself to see their current values where the execution is currently paused. Make sure the editor window is in focus by clicking on it first.

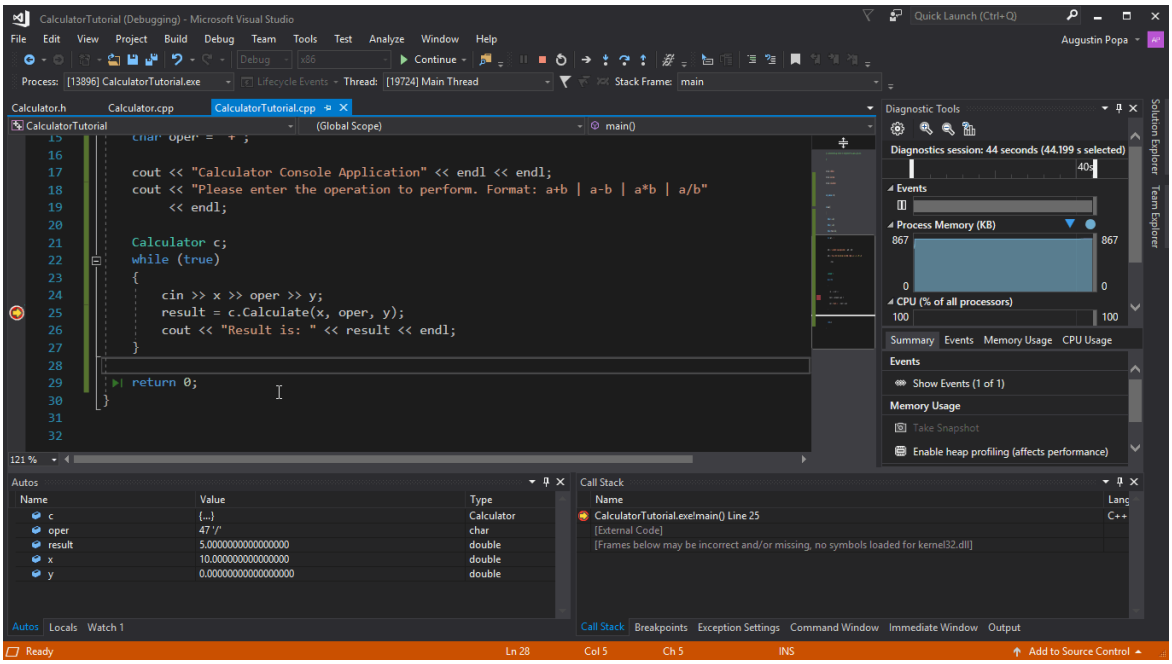


To continue debugging

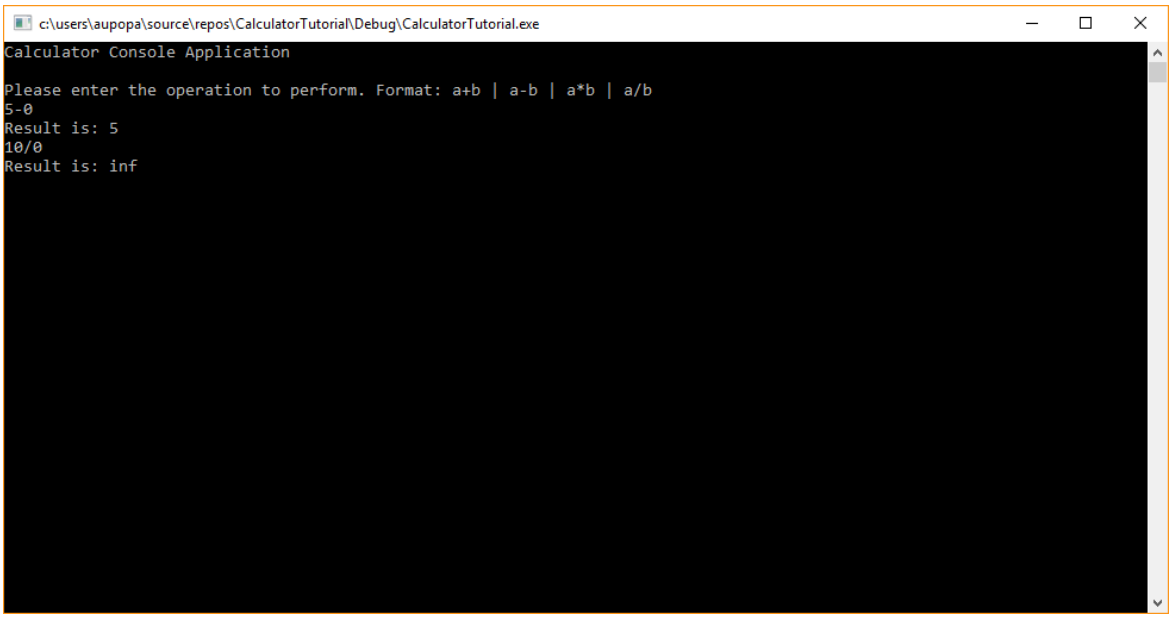
1. The yellow line on the left shows the current point of execution. The current line calls `Calculate`, so press **F11** to **Step Into** the function. You'll find yourself in the body of the `Calculate` function. Be careful with **Step Into**; if you do it too much, you may waste a lot of time. It goes into any code you use on the line you are on, including standard library functions.

2. Now that the point of execution is at the start of the `Calculate` function, press **F10** to move to the next line in the program's execution. **F10** is also known as **Step Over**. You can use **Step Over** to move from line to line, without delving into the details of what is occurring in each part of the line. In general you should use **Step Over** instead of **Step Into**, unless you want to dive more deeply into code that is being called from elsewhere (as you did to reach the body of `Calculate`).

3. Continue using **F10** to **Step Over** each line until you get back to the `main()` function in the other file, and stop on the `cout` line.



It looks like the program is doing what is expected: it takes the first number, and divides it by the second. On the `cout` line, hover over the `result` variable or take a look at `result` in the **Autos** window. You'll see its value is listed as "inf", which doesn't look right, so let's fix it. The `cout` line just outputs whatever value is stored in `result`, so when you step one more line forward using **F10**, the console window displays:



This result happens because division by zero is undefined, so the program doesn't have a numerical answer to the requested operation.

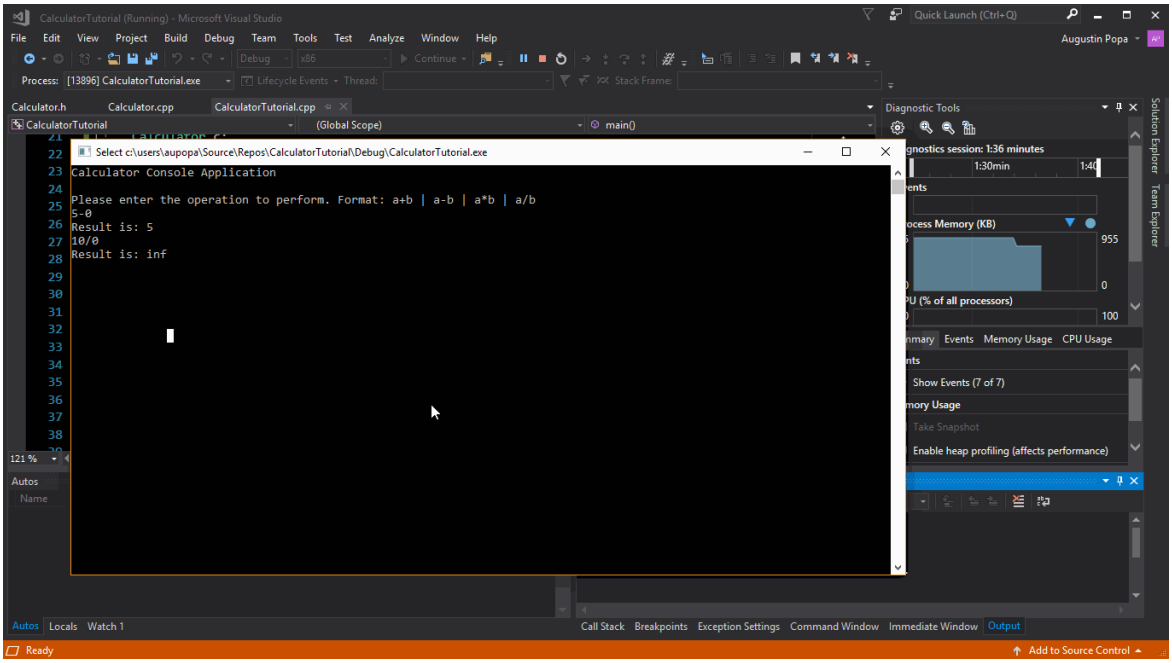
To fix the "divide by zero" error

Let's handle division by zero more gracefully, so a user can understand the problem.

1. Make the following changes in *CalculatorTutorial.cpp*. (You can leave the program running as you edit, thanks to a debugger feature called **Edit and Continue**):

```
// CalculatorTutorial.cpp : This file contains the 'main' function. Program execution begins and ends
there.
//

#include "pch.h"
#include <iostream>
#include "Calculator.h"

using namespace std;

int main()
{
    double x = 0.0;
    double y = 0.0;
    double result = 0.0;
    char oper = '+';

    cout << "Calculator Console Application" << endl << endl;
    cout << "Please enter the operation to perform. Format: a+b | a-b | a*b | a/b" << endl;

    Calculator c;
    while (true)
    {
        cin  >> x  >> oper  >> y;
        if (oper == '/' && y == 0)
        {
            cout << "Division by 0 exception" << endl;
            continue;
        }
        else
        {
            result = c.Calculate(x, oper, y);
        }
        cout << "Result is: " << result << endl;
    }

    return 0;
}
```

2. Now press **F5** once. Program execution continues all the way until it has to pause to ask for user input. Enter `10 / 0` again. Now, a more helpful message is printed. The user is asked for more input, and the program continues executing normally.



NOTE

When you edit code while in debugging mode, there is a risk of code becoming stale. This happens when the debugger is still running your old code, and has not yet updated it with your changes. The debugger pops up a dialog to inform you when this happens. Sometimes, you may need to press **F5** to refresh the code being executed. In particular, if you make a change inside a function while the point of execution is inside that function, you'll need to step out of the function, then back into it again to get the updated code. If that doesn't work for some reason and you see an error message, you can stop debugging by clicking on the red square in the toolbar under the menus at the top of the IDE, then start debugging again by entering **F5** or by choosing the green "play" arrow beside the stop button on the toolbar.

Understanding the Run and Debug shortcuts

- **F5** (or **Debug** > **Start Debugging**) starts a debugging session if one isn't already active, and runs the program until a breakpoint is hit or the program needs user input. If no user input is needed and no breakpoint is available to hit, the program terminates and the console window closes itself when the program finishes running. If you have something like a "Hello World" program to run, use **Ctrl+F5** or set a breakpoint before you enter **F5** to keep the window open.
- **Ctrl+F5** (or **Debug** > **Start Without Debugging**) runs the application without going into debug mode. This is slightly faster than debugging, and the console window stays open after the program finishes executing.
- **F10** (known as **Step Over**) lets you iterate through code, line-by-line, and visualize how the code is run and what variable values are at each step of execution.
- **F11** (known as **Step Into**) works similarly to **Step Over**, except it steps into any functions called on the line of execution. For example, if the line being executed calls a function, pressing **F11** moves the pointer into the body of the function, so you can follow the function's code being run before coming back to the line you started at. Pressing **F10** steps over the function call and just moves to the next line; the function call still happens, but the program doesn't pause to show you what it's doing.

Close the app

- If it's still running, close the console window for the calculator app.

Congratulations! You've completed the code for the calculator app, and built and debugged it in Visual Studio.

Next steps

Learn more about Visual Studio for C++

Creating an MFC Application

9/2/2022 • 3 minutes to read • Edit Online

An MFC application is an executable application for Windows that is based on the Microsoft Foundation Class (MFC) Library. MFC executables generally fall into five types: standard Windows applications, dialog boxes, forms-based applications, Explorer-style applications, and Web browser-style applications. For more information, see:

- Using the Classes to Write Windows Applications
- Creating and Displaying Dialog Boxes
- Creating a Forms-Based MFC Application
- Creating a File Explorer-Style MFC Application
- Creating a Web Browser-Style MFC Application

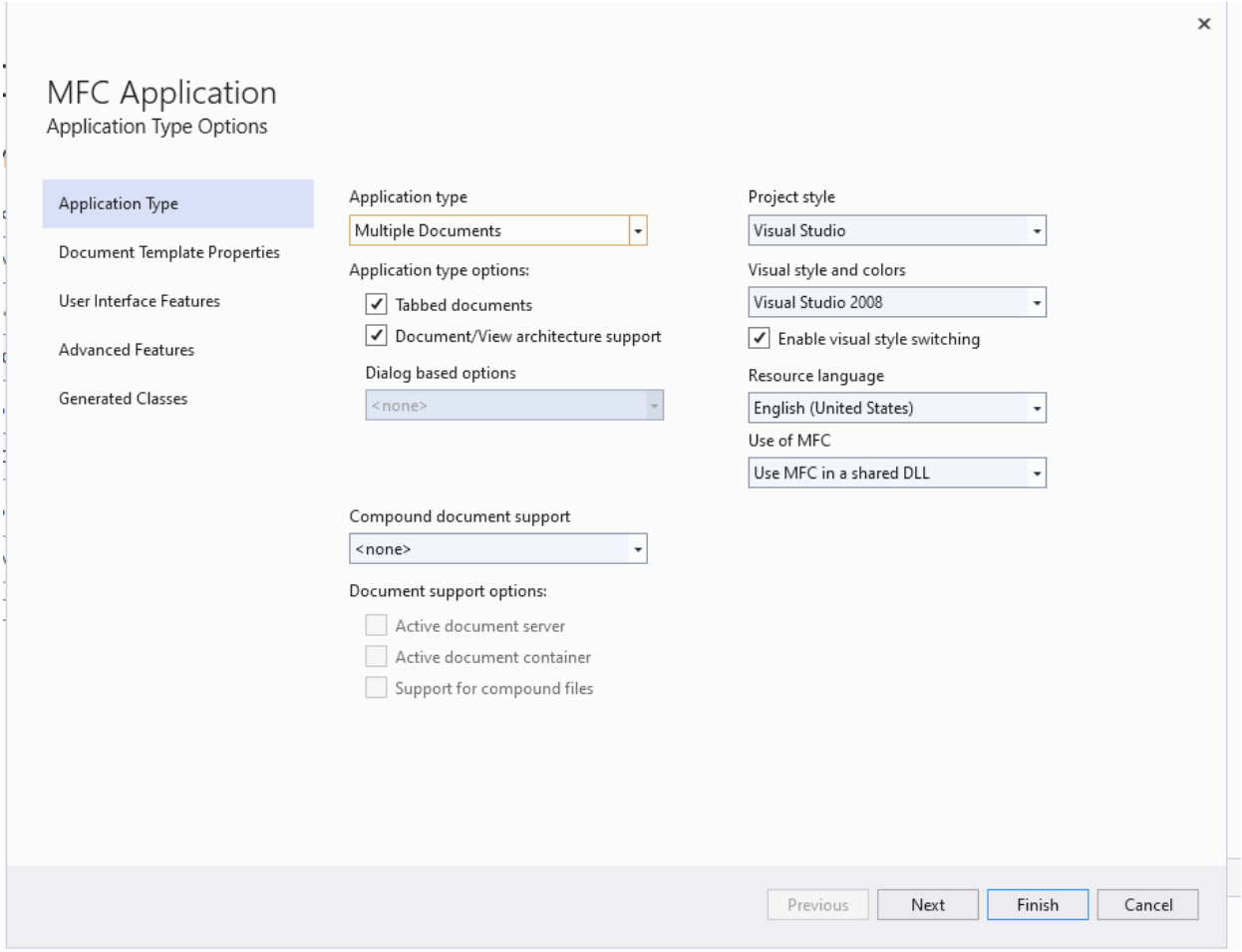
The MFC Application Wizard generates the appropriate classes and files for any of these types of applications, depending on the options you select in the wizard.

The easiest way to create an MFC application is to use the MFC Application Wizard (**MFC App project** in Visual Studio 2019). To create an MFC console application (a command-line program that uses MFC libraries but runs in the console window), use the Windows Desktop Wizard and choose the **Console Application** and **MFC Headers** options.

To create an MFC forms or dialog-based application

1. From the main menu, choose **File** > **New** > **Project**.
2. Enter "MFC" into the search box and then choose **MFC App** from the result list.
3. Modify the defaults as needed, then press **Create** to open the **MFC Application Wizard**.
4. Modify the configuration values as needed, then press **Finish**.

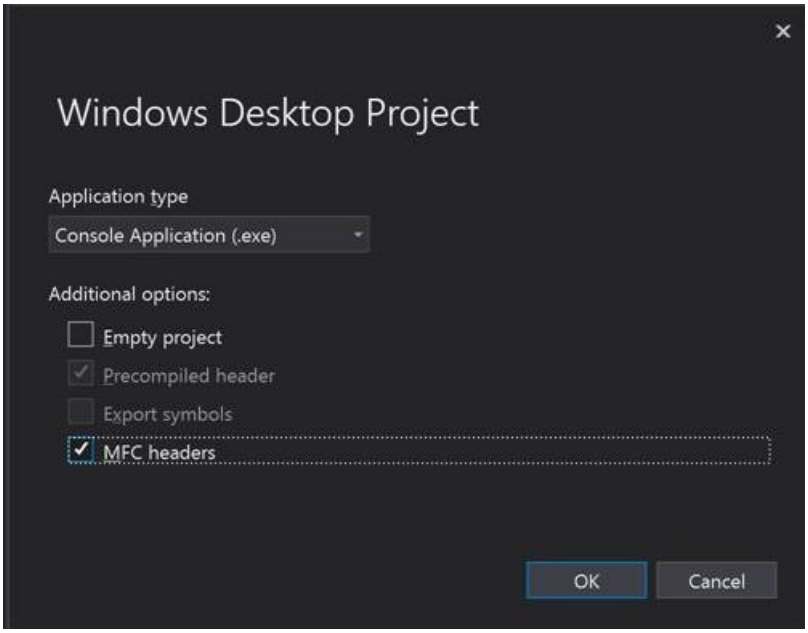
For more information, see Creating a Forms-Based MFC Application.



To create an MFC console application

An MFC console application is a command-line program that uses MFC libraries but runs in the console window.

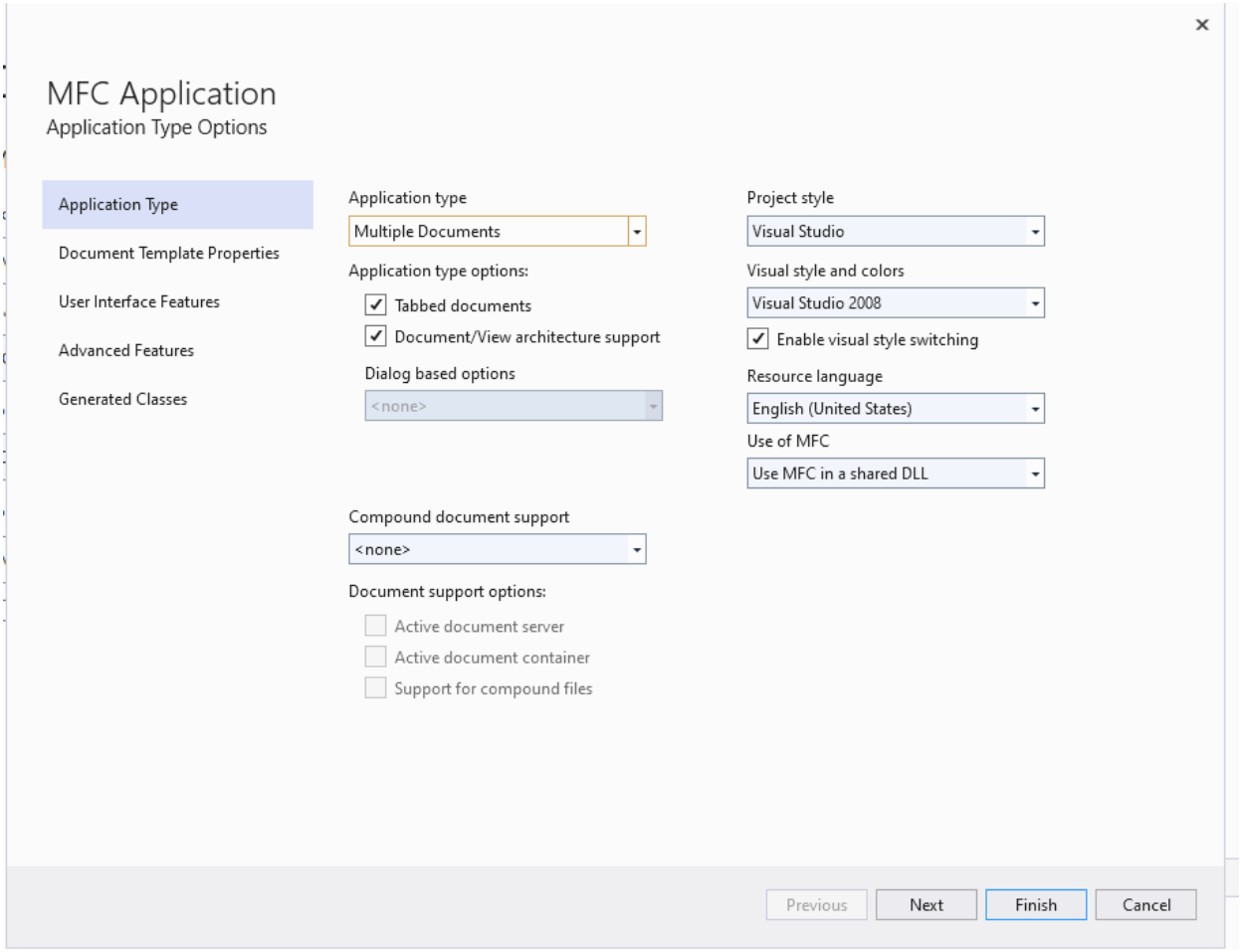
1. From the main menu, choose **File** > **New** > **Project**.
2. Enter "Desktop" into the search box and then choose **Windows Desktop Wizard** from the result list.
3. Modify the project name as needed, then press **Next** to open the **Windows Desktop Wizard**.
4. Check the **MFC Headers** box and set other values as needed, then press **Finish**.



To create an MFC forms or dialog-based application

1. From the main menu, choose **File** > **New** > **Project**.
2. Under the **Installed** templates, choose **Visual C++** > **MFC/ATL**. If you don't see these, use the Visual Studio Installer to add them.
3. Choose **MFC Application** from the center pane.
4. Modify the configuration values as needed, then press **Finish**.

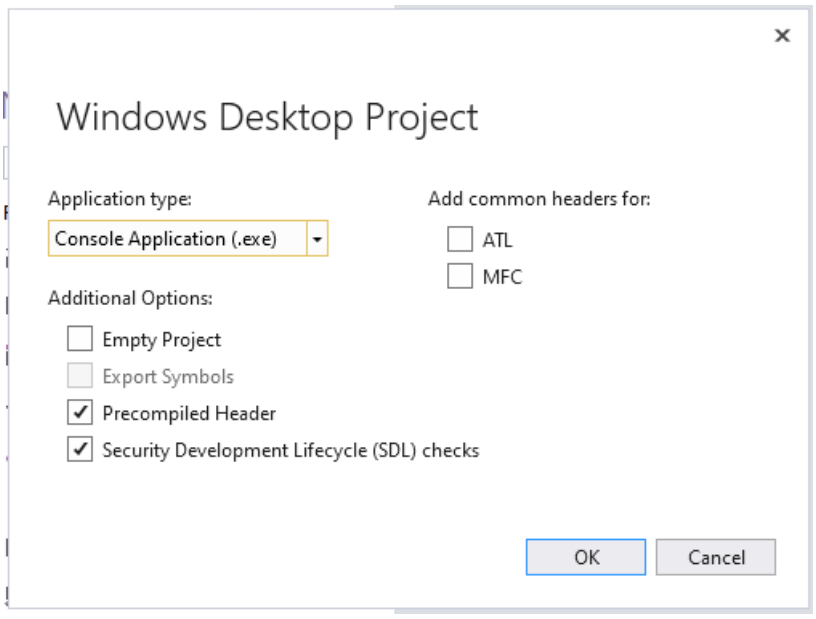
For more information, see Creating a Forms-Based MFC Application.



To create an MFC console application

An MFC console application is a command-line program that uses MFC libraries but runs in the console window.

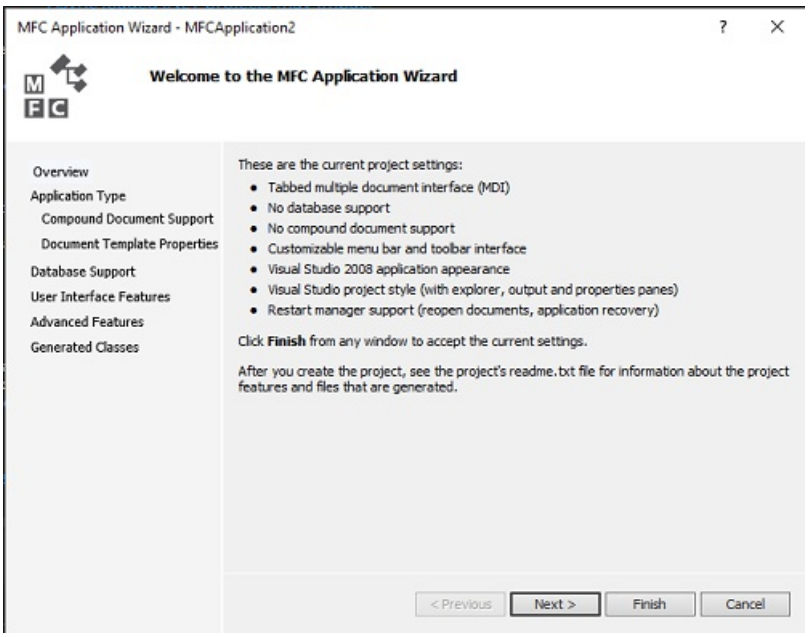
1. From the main menu, choose **File** > **New** > **Project**.
2. Under the **Installed** templates, choose **Visual C++** > **Windows Desktop**.
3. Choose **Windows Desktop Wizard** from the center pane.
4. Modify the project name as needed, then press **OK** to open the **Windows Desktop Wizard**.
5. Check the **MFC Headers** box and set other values as needed, then press **Finish**.



To create an MFC forms or dialog-based application

1. From the main menu, choose **File** > **New** > **Project**.
2. Under the **Installed** templates, choose **Visual C++** > **MFC**.
3. Choose **MFC Application** from the center pane.
4. Click **Next** to start the **MFC Application Wizard**.

For more information, see Creating a Forms-Based MFC Application.



To create an MFC console application

An MFC console application is a command-line program that uses MFC libraries but runs in the console window.

1. From the main menu, choose **File** > **New** > **Project**.
2. Under the **Installed** templates, choose **Visual C++** > **Win32**.
3. Choose **Win32 Console Application** from the center pane.
4. Modify the project name as needed, then press **OK**.
5. On the second page of the wizard, check the **Add common headers for MFC** box and set other values as needed, then press **Finish**.

Once your project is created, you can view the files created in **Solution Explorer**. For more information about the files the wizard creates for your project, see the project-generated file ReadMe.txt. For more information about the file types, see File Types Created for Visual Studio C++ projects.

See also

Adding Functionality with Code Wizards
Property Pages

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

9/2/2022 • 23 minutes to read • Edit Online

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.

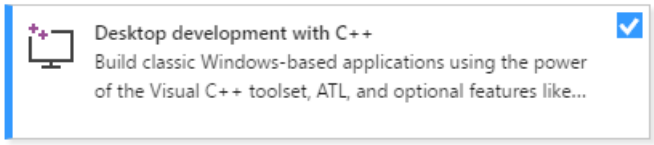
Even though the code of the DLL is written in C++, we've used C-style interfaces for the exported functions. There are two main reasons for this: First, many other languages support imports of C-style functions. The client app doesn't have to be written in C++. Second, it avoids some common pitfalls related to exported classes and member functions. It's easy to make hard-to-diagnose errors when exporting classes, since everything referred to within a class declaration has to have an instantiation that's also exported. This restriction applies to DLLs, but not static libraries. If your classes are plain-old-data style, you shouldn't run into this issue.

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.



- A copy of Visual Studio. For information on how to download and install Visual Studio 2015, see Install Visual Studio 2015. Use a **Custom** installation to install the C++ compiler and tools, since they're not installed by default.
- 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.

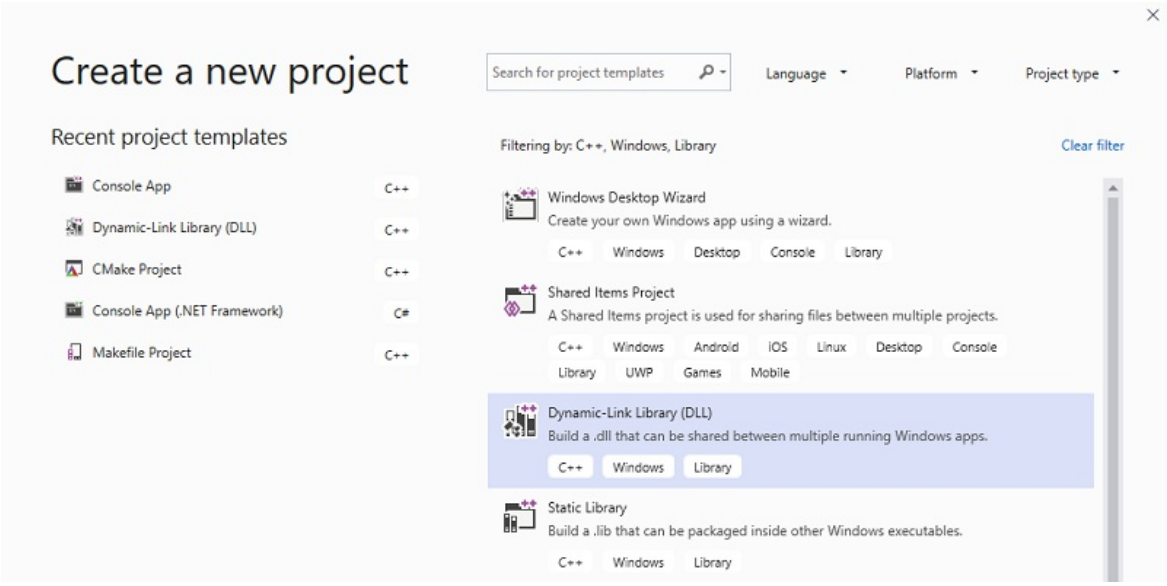
> **NOTE**
> This walkthrough assumes you're using Visual Studio 2017 version 15.9 or later. Some earlier versions of Visual Studio 2017 had defects in the code templates, or used different user interface dialogs. To avoid problems, use the Visual Studio Installer to update Visual Studio 2017 to version 15.9 or later.

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.



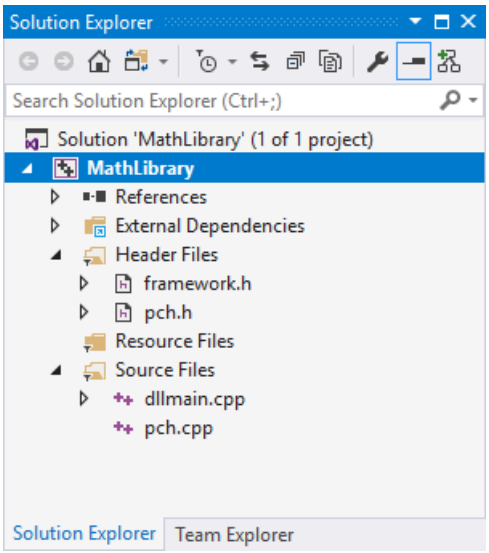
2. At the top of the dialog, set **Language** to **C++**, set **Platform** to **Windows**, and set **Project type** to **Library**.

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.

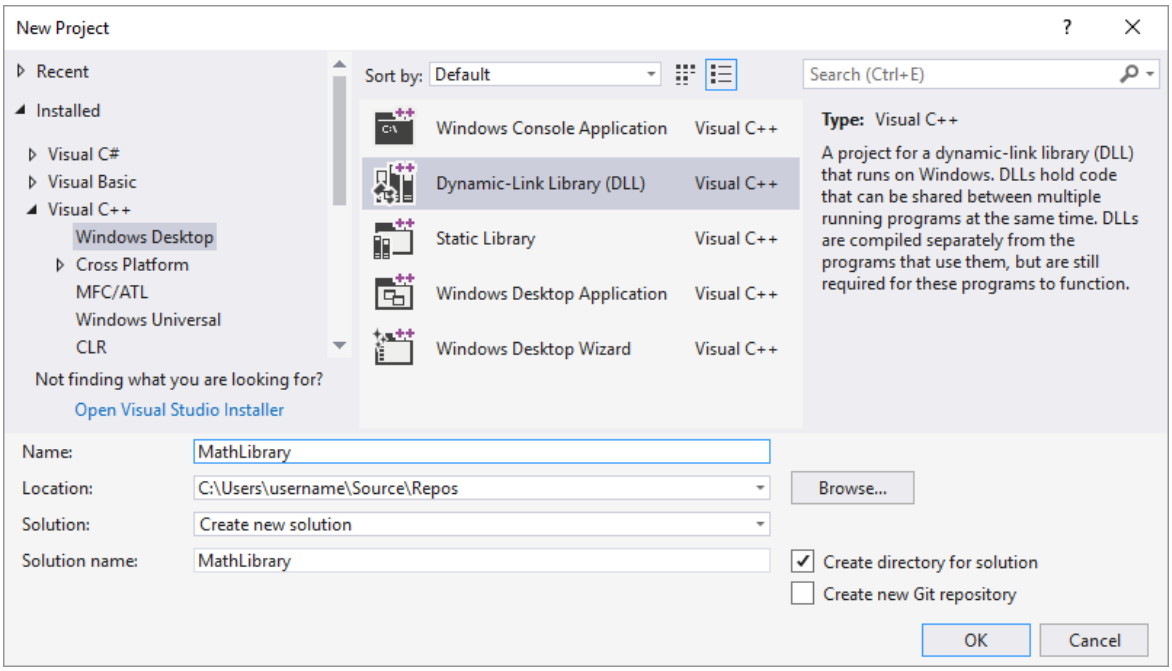
When the solution is created, you can see the generated project and source files in the **Solution Explorer** window in Visual Studio.



To create a DLL project in Visual Studio 2017

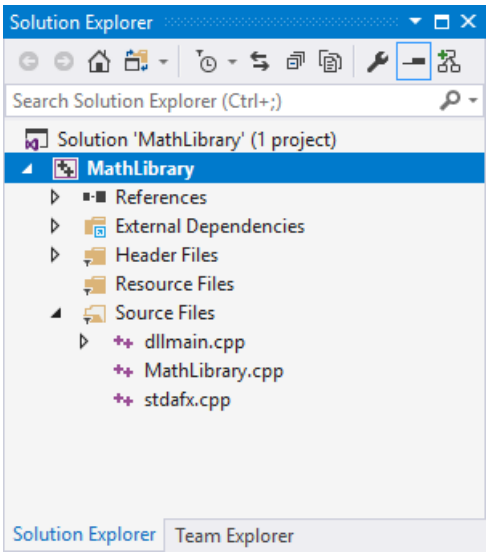
1. On the menu bar, choose **File** > **New** > **Project** to open the **New Project** dialog box.

2. In the left pane of the **New Project** dialog box, select **Installed** > **Visual C++** > **Windows Desktop**. In the center pane, select **Dynamic-Link Library (DLL)**. Enter *MathLibrary* in the **Name** box to specify a name for the project. Leave the default **Location** and **Solution name** values. Set **Solution** to **Create new solution**. Check **Create directory for solution** if it's unchecked.



3. Choose the **OK** button to create the project.

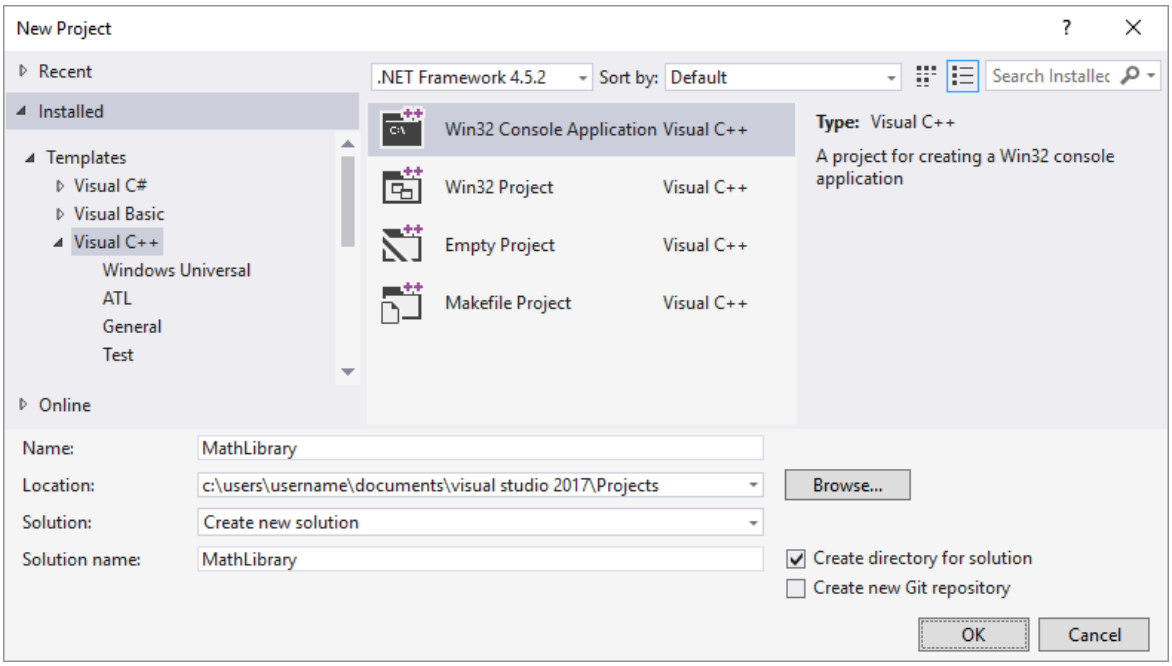
When the solution is created, you can see the generated project and source files in the **Solution Explorer** window in Visual Studio.



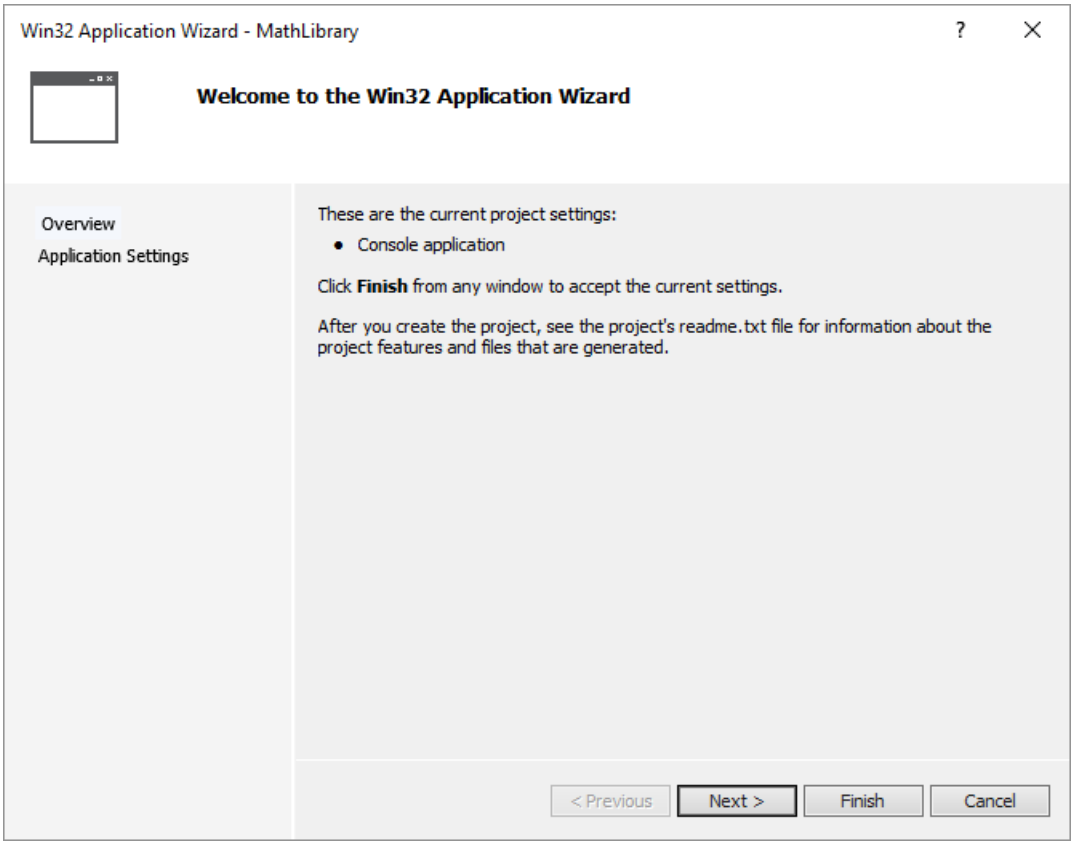
To create a DLL project in Visual Studio 2015 and older versions

1. On the menu bar, choose **File** > **New** > **Project**.

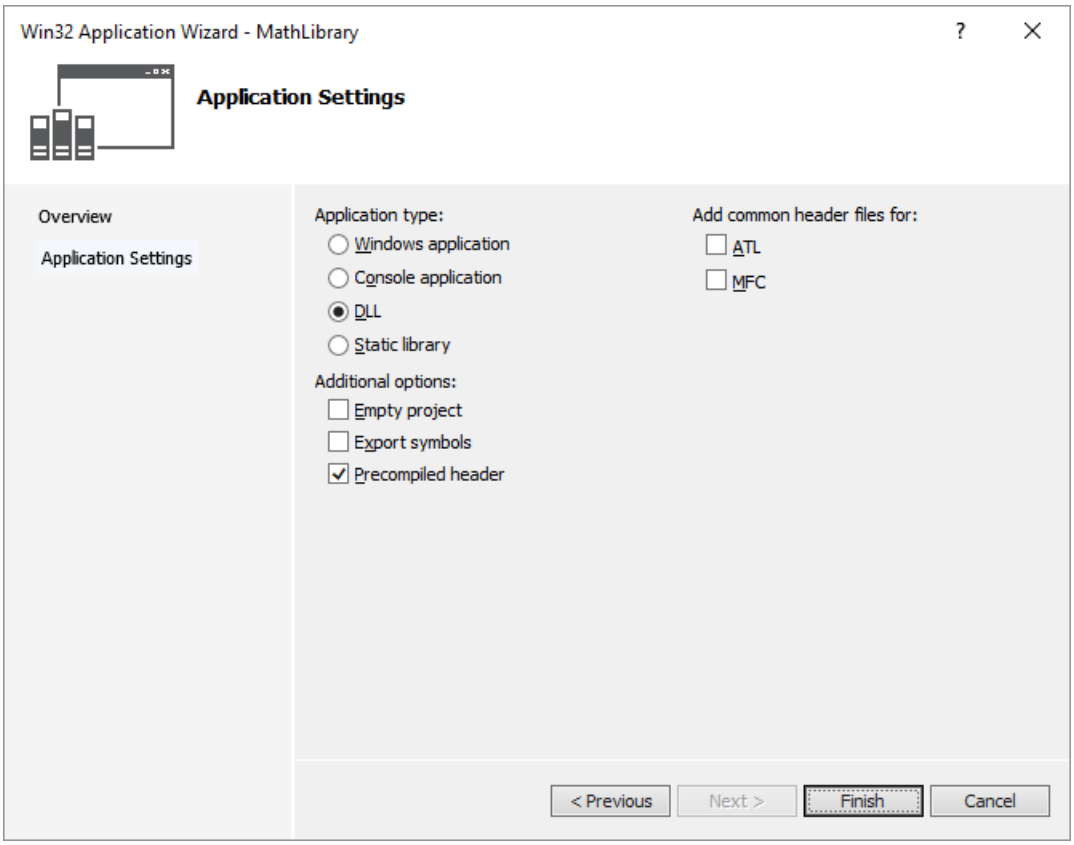
2. In the left pane of the **New Project** dialog box, expand **Installed** > **Templates**, and select **Visual C++**, and then in the center pane, select **Win32 Console Application**. Enter *MathLibrary* in the **Name** edit box to specify a name for the project. Leave the default **Location** and **Solution name** values. Set **Solution** to **Create new solution**. Check **Create directory for solution** if it's unchecked.



3. Choose the **OK** button to dismiss the **New Project** dialog and start the **Win32 Application Wizard**.

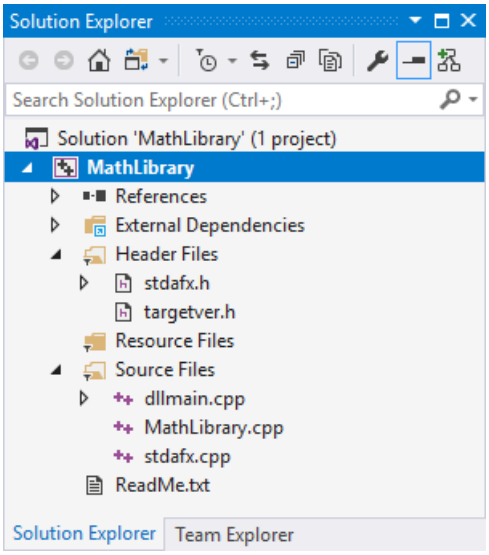


4. Choose the **Next** button. On the **Application Settings** page, under **Application type**, select **DLL**.



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

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

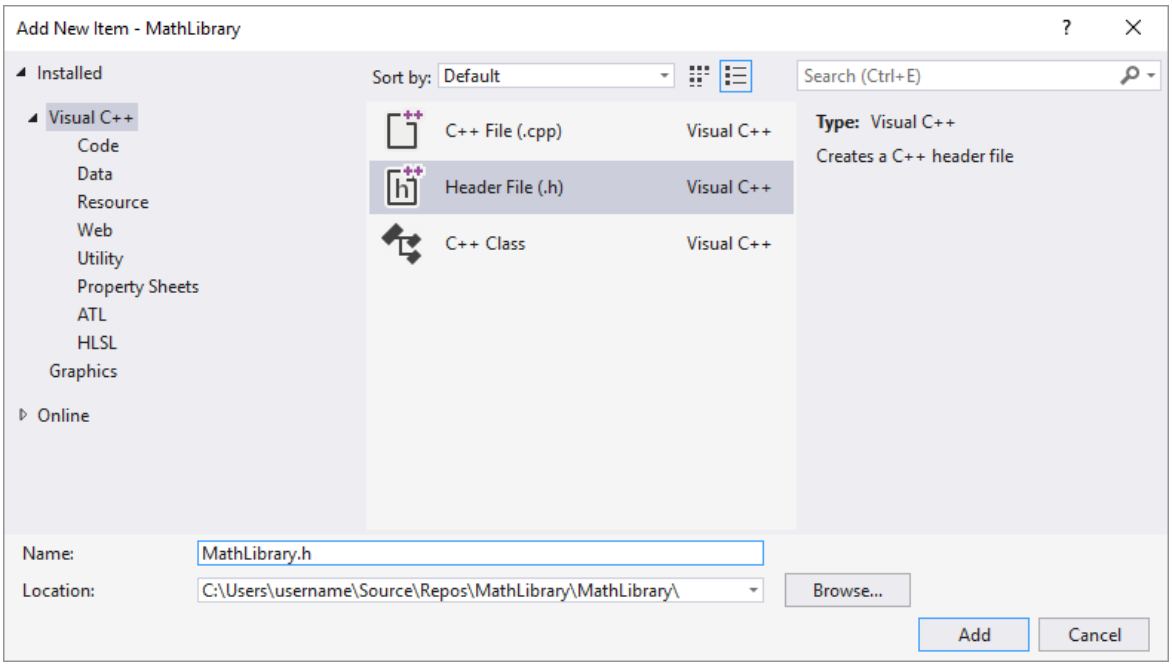


Right now, this DLL doesn't do very much. Next, you'll create a header file to declare the functions your DLL exports, and then add the function definitions to the DLL to make it more useful.

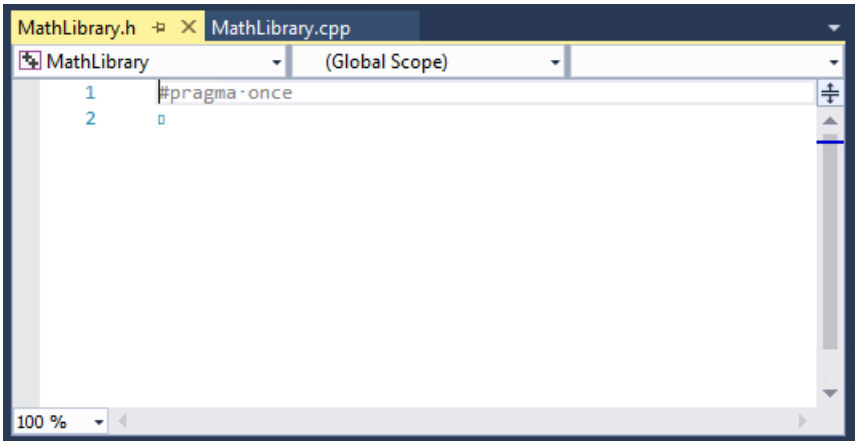
To add a header file to the DLL

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.



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



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

```
// 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)
// 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:

```
// 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_;
}
```

1. 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.

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

```
// MathLibrary.cpp : Defines the exported functions for the DLL.
#include "stdafx.h" // use pch.h in Visual Studio 2019 and later
#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:

```
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 ==========
```

```
1>------ Build started: Project: MathLibrary, Configuration: Debug Win32 ------
1>stdafx.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 ==========
```

```
1>------ Build started: Project: MathLibrary, Configuration: Debug Win32 ------
1>MathLibrary.cpp
1>dllmain.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
1>MathLibrary.vcxproj -> C:\Users\username\Source\Repos\MathLibrary\Debug\MathLibrary.pdb (Partial PDB)
========== 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.

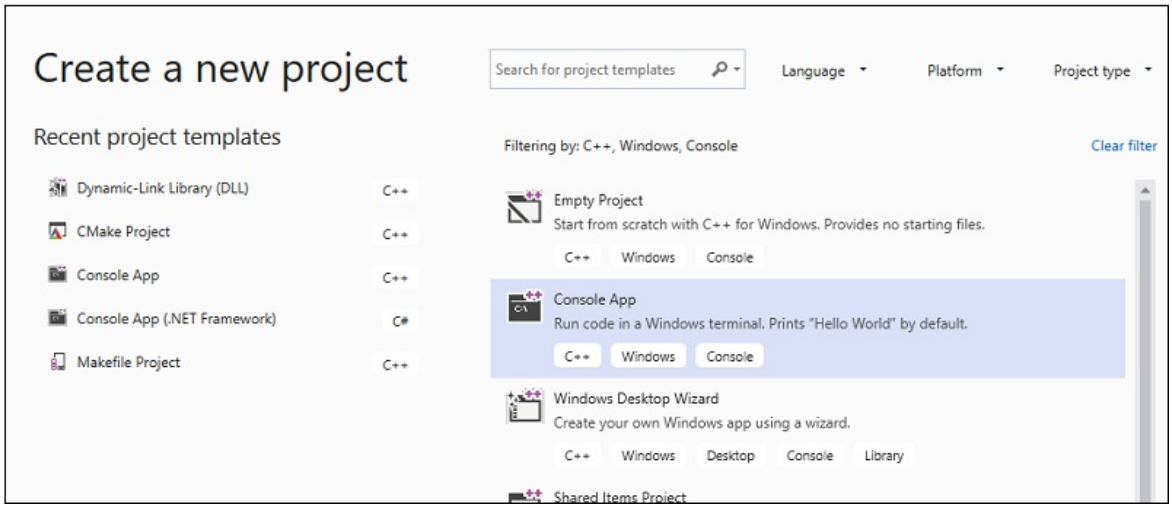
To create a client app in Visual Studio

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.



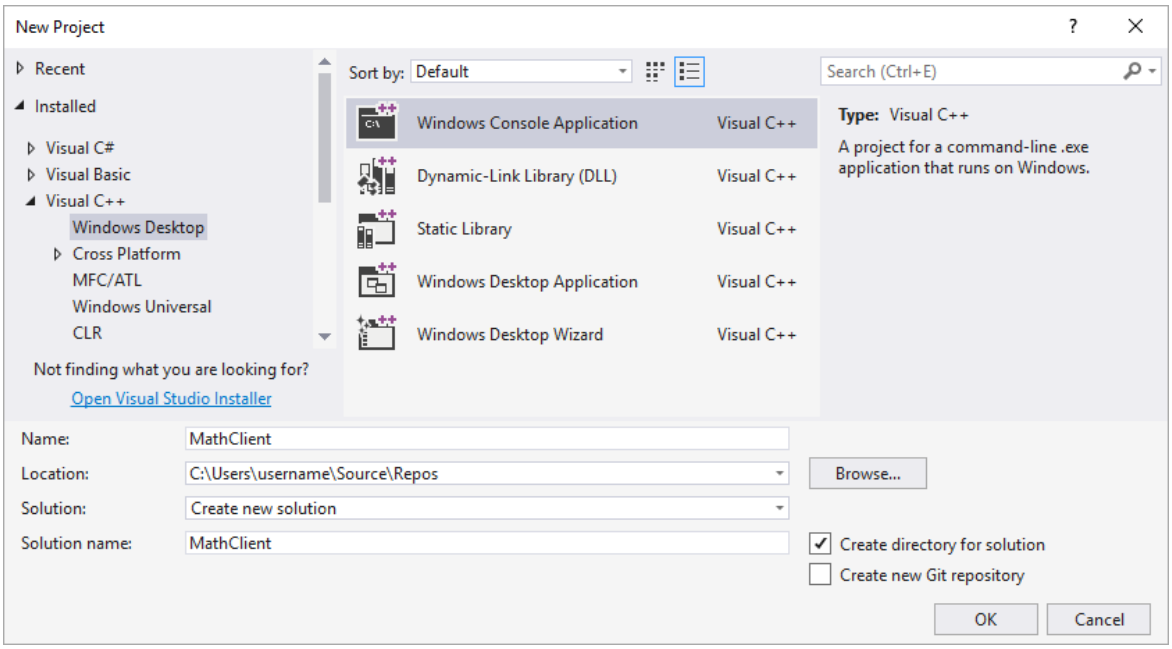
5. Choose the **Create** button to create the client project.

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.

To create a client app in Visual Studio 2017

1. To create a C++ app that uses the DLL that you created, on the menu bar, choose **File** > **New** > **Project**.

2. In the left pane of the **New Project** dialog, select **Windows Desktop** under **Installed** > **Visual C++**. In the center pane, select **Windows Console Application**. Specify the name for the project, *MathClient*, in the **Name** edit box. Leave the default **Location** and **Solution name** values. Set **Solution** to **Create new solution**. Check **Create directory for solution** if it's unchecked.

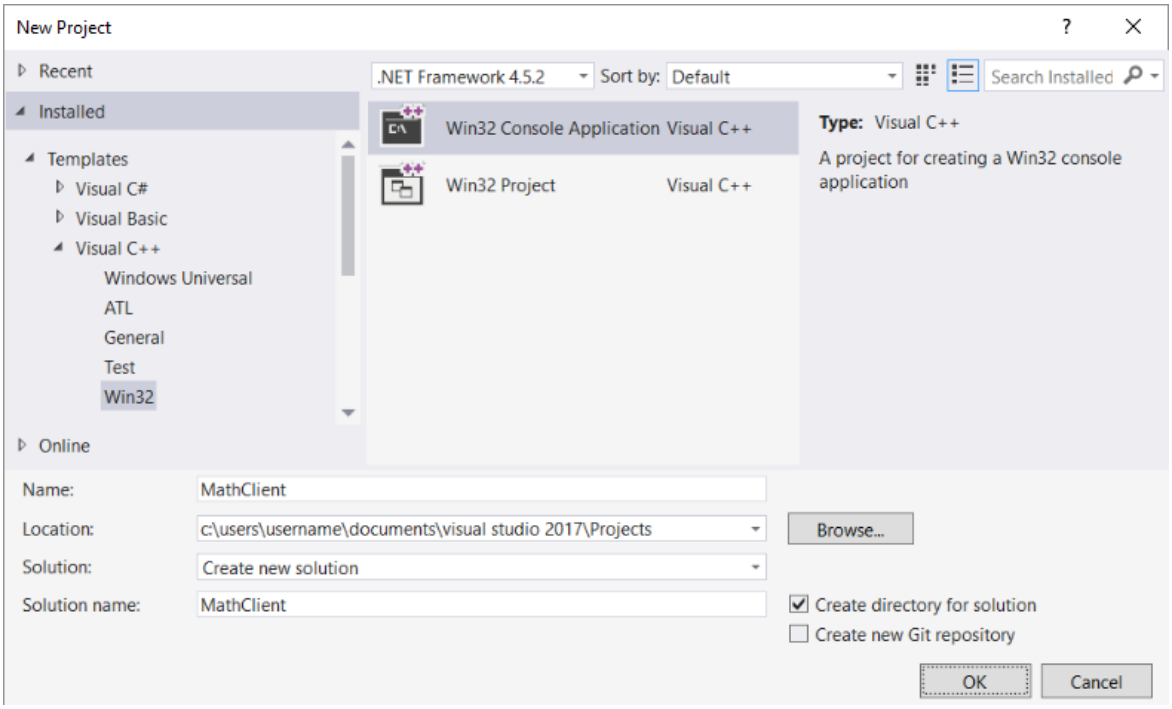


3. Choose **OK** to create the client app project.

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.

To create a client app in Visual Studio 2015

1. To create a C++ app that uses the DLL that you created, on the menu bar, choose **File** > **New** > **Project**.
2. In the left pane of the **New Project** dialog, select **Win32** under **Installed** > **Templates** > **Visual C++**. In the center pane, select **Win32 Console Application**. Specify the name for the project, *MathClient*, in the **Name** edit box. Leave the default **Location** and **Solution name** values. Set **Solution** to **Create new solution**. Check **Create directory for solution** if it's unchecked.



3. Choose the **OK** button to dismiss the **New Project** dialog and start the **Win32 Application Wizard**. On the **Overview** page of the **Win32 Application Wizard** dialog box, choose the **Next** button.
4. On the **Application Settings** page, under **Application type**, select **Console application** if it isn't already selected.
5. Choose the **Finish** button to create the project.

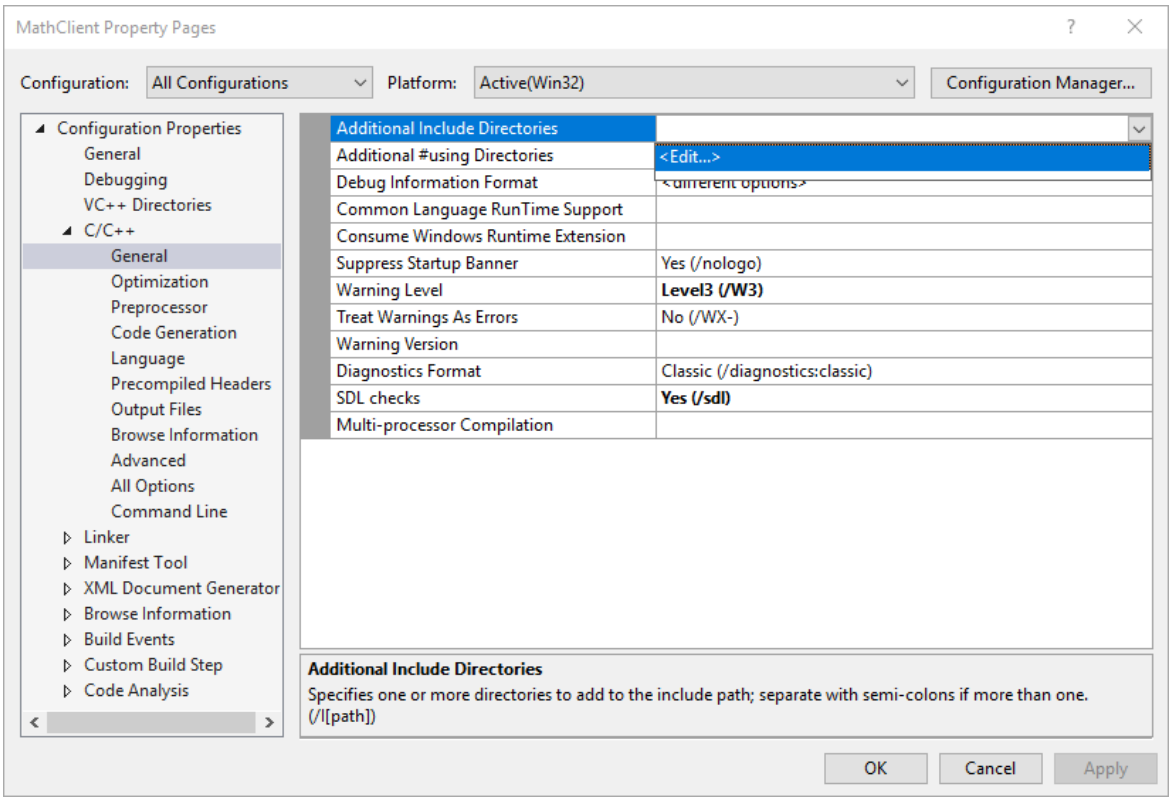
When the wizard finishes, a minimal console application project is created for you. The name for the main source file is the same as the project name that you entered earlier. In this example, it's named **MathClient.cpp**. You can build it, but it doesn't use your DLL yet.

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**.



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.

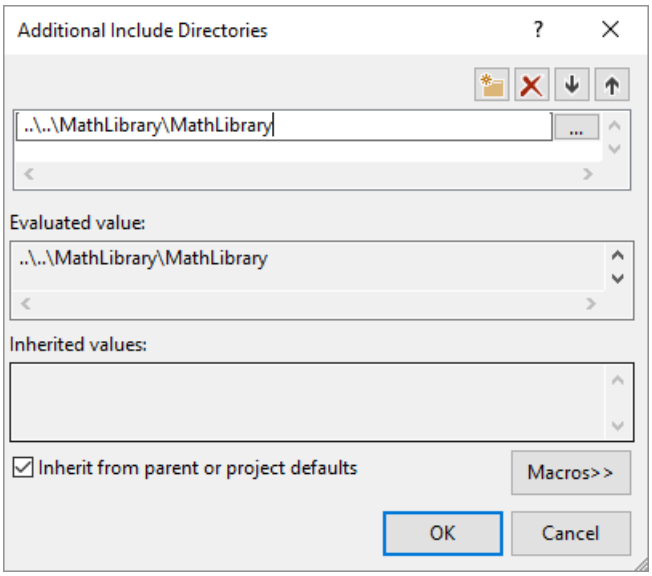
   You can also enter a relative path from your client source files to the folder that contains the DLL header files. If you followed the directions to put your client project in a separate solution from the DLL, the relative path should look like this:

   ```
   ..\..\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.



7. After you've entered the path to the header file in the **Additional Include Directories** dialog box, choose the **OK** button. In the **Property Pages** dialog box, choose the **OK** button to save your changes.

You can now include the **MathLibrary.h** file and use the functions it declares in your client application. Replace the contents of **MathClient.cpp** by using this code:

```
// MathClient.cpp : Client app for MathLibrary DLL.
// #include "pch.h" Uncomment for Visual Studio 2017 and earlier
#include <iostream>
#include "MathLibrary.h"

int main()
{
    // Initialize a Fibonacci relation sequence.
    fibonacci_init(1, 1);
    // Write out the sequence values until overflow.
    do {
        std::cout << fibonacci_index() << ": "
            << fibonacci_current() << std::endl;
    } while (fibonacci_next());
    // Report count of values written before overflow.
    std::cout << fibonacci_index() + 1 <<
        " Fibonacci sequence values fit in an " <<
        "unsigned 64-bit integer." << std::endl;
}
```

This code can be compiled, but not linked. If you build the client app now, the error list shows several LNK2019 errors. That's because your project is missing some information: You haven't specified that your project has a dependency on the *MathLibrary.lib* library yet. And, you haven't told the linker how to find the *MathLibrary.lib* file.

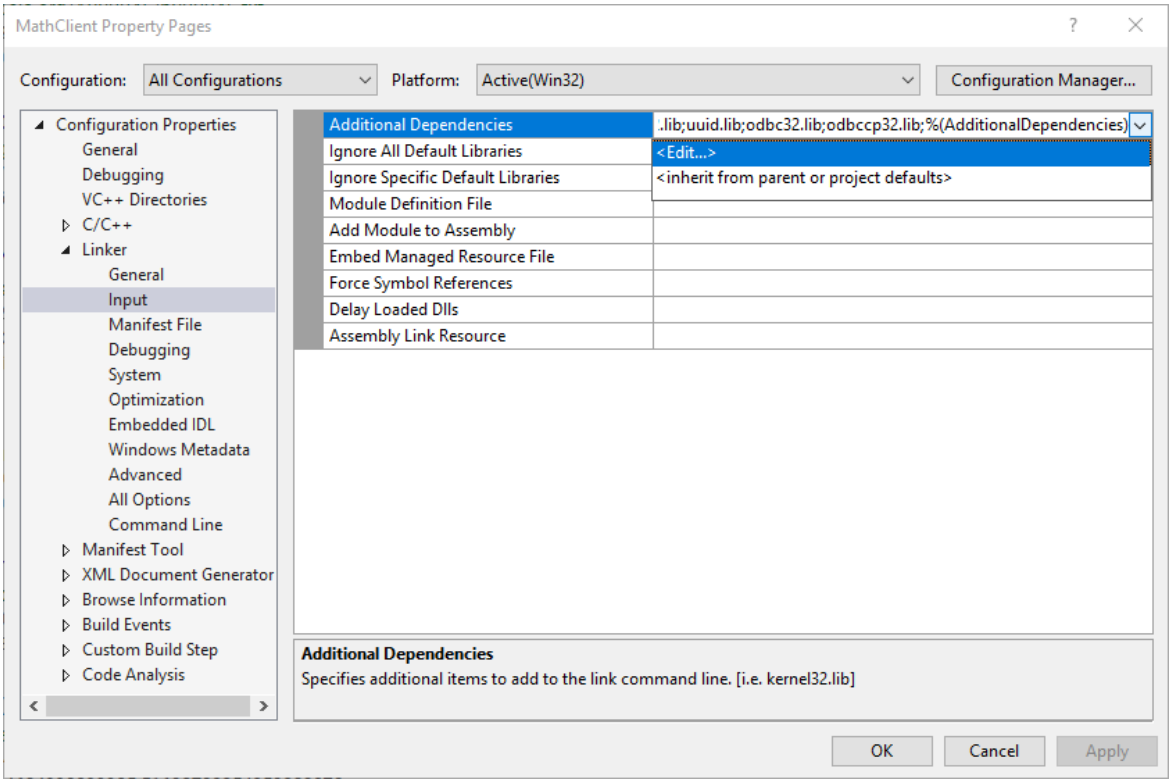
To fix this issue, you could copy the library file directly into your client app project. The linker would find and use it automatically. However, if both the library and the client app are under development, that might lead to changes in one copy that aren't shown in the other. To avoid this issue, you can set the **Additional Dependencies** property to tell the build system that your project depends on *MathLibrary.lib*. And, you can set an **Additional Library Directories** path in your project to include the path to the original library when you link.

To add the DLL import library to your project

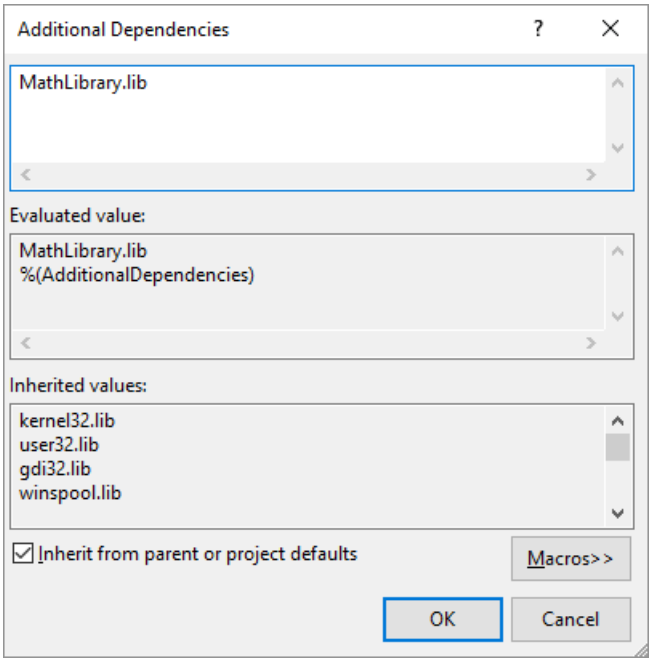
1. Right-click on the **MathClient** node in **Solution Explorer** and choose **Properties** to open the **Property Pages** dialog.

2. In the **Configuration** drop-down box, select **All Configurations** if it's not already selected. It ensures that any property changes apply to both Debug and Release builds.

3. In the left pane, select **Configuration Properties** > **Linker** > **Input**. In the property pane, select the drop-down control next to the **Additional Dependencies** edit box, and then choose **Edit**.

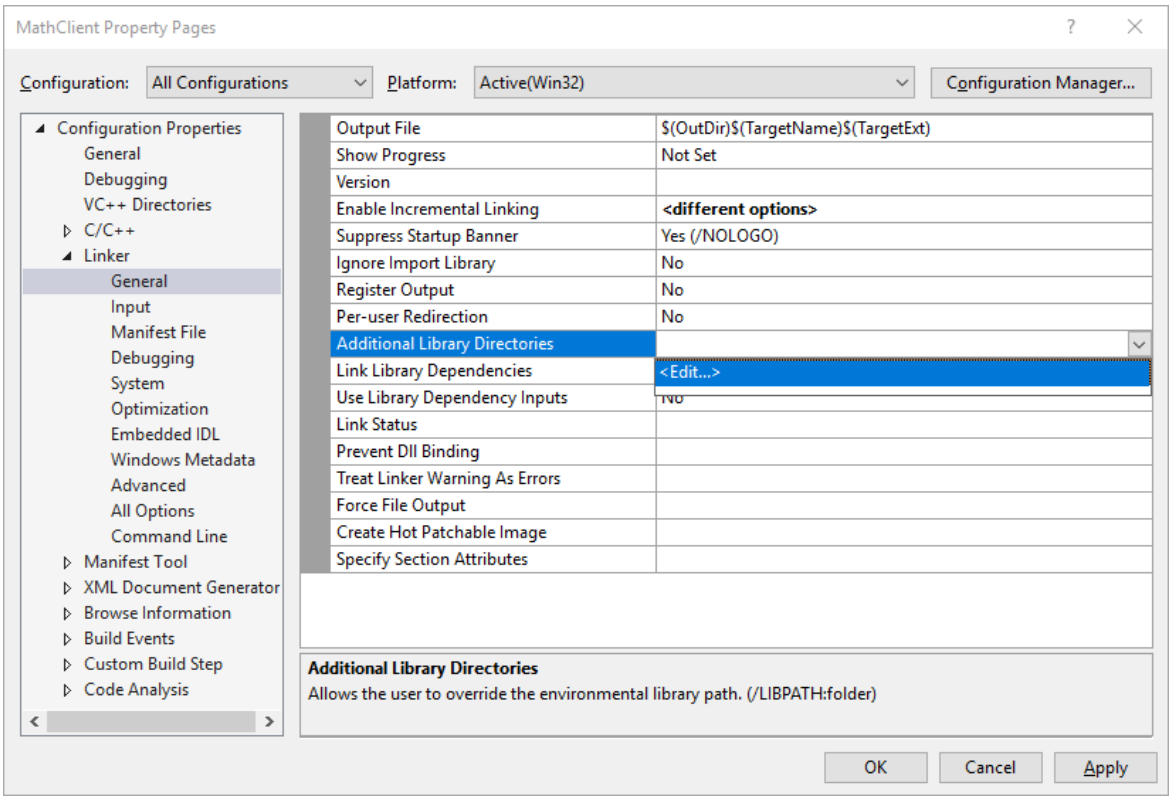


4. In the **Additional Dependencies** dialog, add *MathLibrary.lib* to the list in the top edit control.



5. Choose **OK** to go back to the **Property Pages** dialog box.

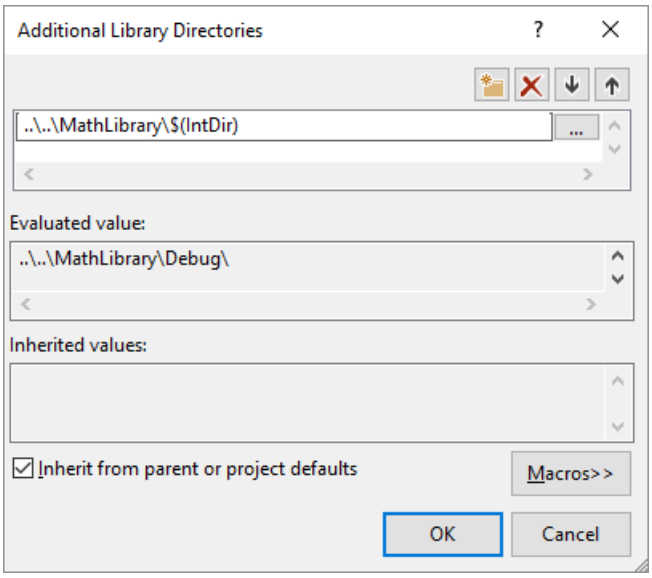
6. In the left pane, select **Configuration Properties** > **Linker** > **General**. In the property pane, select the drop-down control next to the **Additional Library Directories** edit box, and then choose **Edit**.



7. Double-click in the top pane of the **Additional Library Directories** dialog box to enable an edit control. In the edit control, specify the path to the location of the **MathLibrary.lib** file. By default, it's in a folder called *Debug* directly under the DLL solution folder. If you create a release build, the file is placed in a folder called *Release*. You can use the `$(IntDir)` macro so that the linker can find your DLL, no matter which kind of build you create. If you followed the directions to put your client project in a separate solution from the DLL project, the relative path should look like this:

   ```
   ..\..\MathLibrary\$(IntDir)
   ```

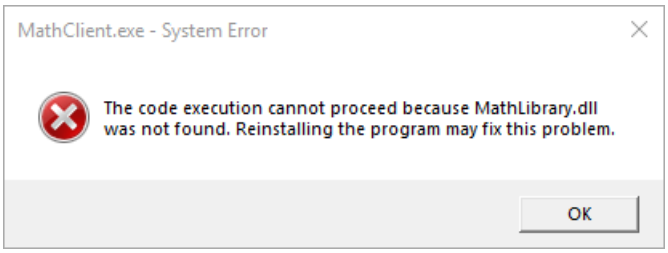
   If your DLL and client projects are in other locations, adjust the relative path to match.



8. Once you've entered the path to the library file in the **Additional Library Directories** dialog box, choose the **OK** button to go back to the **Property Pages** dialog box. Choose **OK** to save the property changes.

Your client app can now compile and link successfully, but it still doesn't have everything it needs to run. When the operating system loads your app, it looks for the MathLibrary DLL. If it can't find the DLL in certain system directories, the environment path, or the local app directory, the load fails. Depending on the operating system,

you'll see an error message like this:



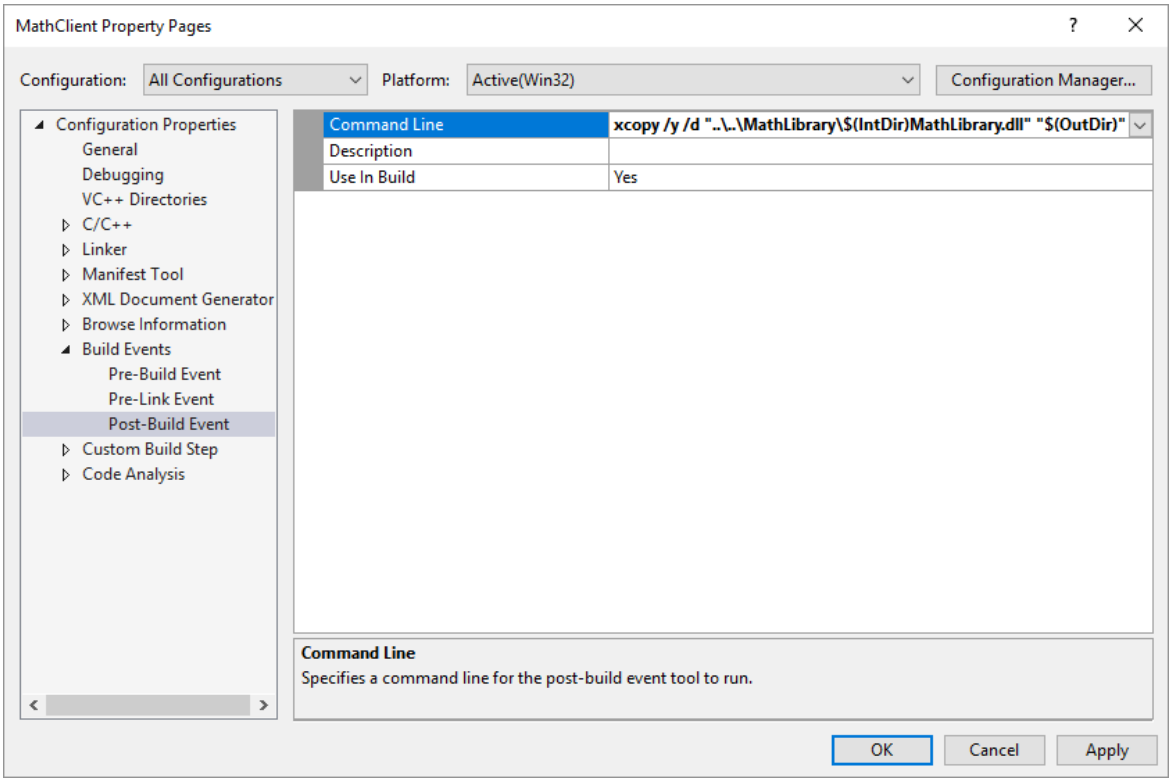
One way to avoid this issue is to copy the DLL to the directory that contains your client executable as part of the build process. You can add a **Post-Build Event** to your project, to add a command that copies the DLL to your build output directory. The command specified here copies the DLL only if it's missing or has changed. It uses macros to copy to and from the Debug or Release locations, based on your build configuration.

To copy the DLL in a post-build event

1. Right-click on the **MathClient** node in **Solution Explorer** and choose **Properties** to open the **Property Pages** dialog.
2. In the **Configuration** drop-down box, select **All Configurations** if it isn't already selected.
3. In the left pane, select **Configuration Properties** > **Build Events** > **Post-Build Event**.
4. In the property pane, select the edit control in the **Command Line** field. If you followed the directions to put your client project in a separate solution from the DLL project, then enter this command:

   ```
   xcopy /y /d "..\..\MathLibrary\$(IntDir)MathLibrary.dll" "$(OutDir)"
   ```

   If your DLL and client projects are in other directories, change the relative path to the DLL to match.

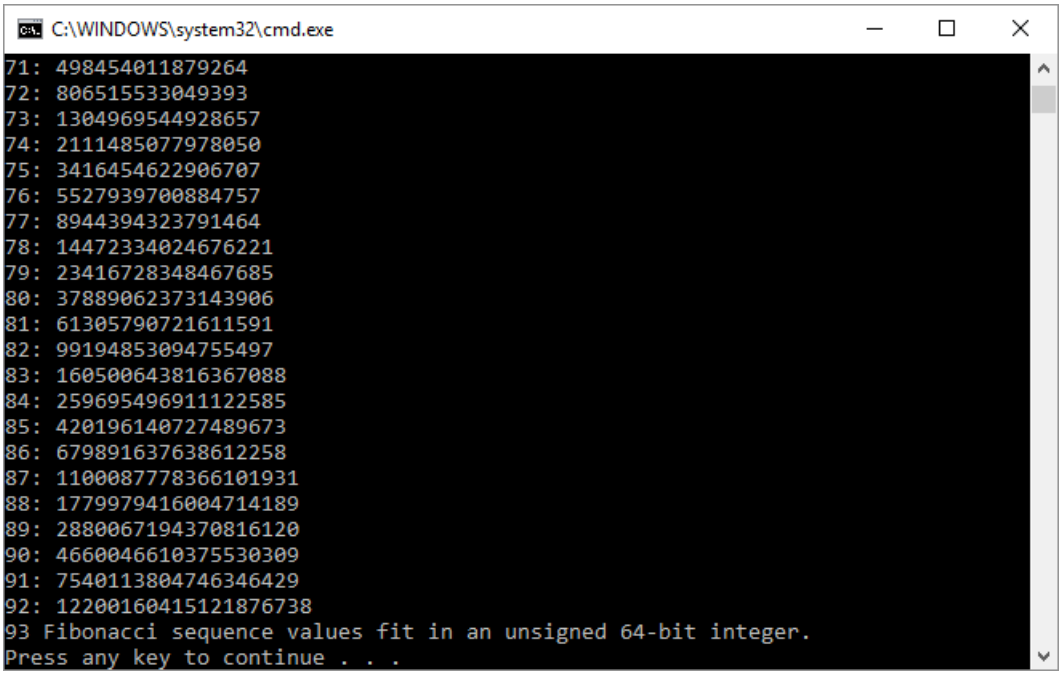


5. Choose the **OK** button to save your changes to the project properties.

Now your client app has everything it needs to build and run. Build the application by choosing **Build** > **Build Solution** on the menu bar. The **Output** window in Visual Studio should have something like the following example depending on your version of Visual Studio:

```
1>------ Build started: Project: MathClient, Configuration: Debug Win32 ------
1>MathClient.cpp
1>MathClient.vcxproj -> C:\Users\username\Source\Repos\MathClient\Debug\MathClient.exe
1>1 File(s) copied
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Congratulations, you've created an application that calls functions in your DLL. Now run your application to see what it does. On the menu bar, choose **Debug** > **Start Without Debugging**. Visual Studio opens a command window for the program to run in. The last part of the output should look like:



Press any key to dismiss the command window.

Now that you've created a DLL and a client application, you can experiment. Try setting breakpoints in the code of the client app, and run the app in the debugger. See what happens when you step into a library call. Add other functions to the library, or write another client app that uses your DLL.

When you deploy your app, you must also deploy the DLLs it uses. The simplest way to make the DLLs that you build, or that you include from third parties, available is to put them in the same directory as your app. It's known as *app-local deployment*. For more information about deployment, see Deployment in Visual C++.

See also

Calling DLL Functions from Visual Basic Applications

Walkthrough: Create and use a static library

9/2/2022 • 9 minutes to read • Edit Online

This step-by-step walkthrough shows how to create a static library (.lib file) for use with C++ apps. Using a static library is a great way to reuse code. Rather than reimplementing the same routines in every app that requires the functionality, you write them one time in a static library and then reference it from the apps. Code linked from a static library becomes part of your app—you don't have to install another file to use the code.

This walkthrough covers these tasks:

- Create a static library project
- Add a class to the static library
- Create a C++ console app that references the static library
- Use the functionality from the static library in the app
- Run the app

Prerequisites

An understanding of the fundamentals of the C++ language.

Create a static library project

The instructions for how to create the project vary depending on your version of Visual Studio. To see the documentation for your preferred version of Visual Studio, use the **Version** selector control. It's found at the top of the table of contents on this page.

To create a static library project in Visual Studio

1. On the menu bar, choose **File** > **New** > **Project** to open the **Create a New Project** dialog.
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 **Windows Desktop Wizard**, 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. Enter *StaticMath* in the **Solution name** box. Choose the **Create** button to open the **Windows Desktop Project** dialog.
5. In the **Windows Desktop Project** dialog, under **Application type**, select **Static Library (.lib)**.
6. Under **Additional options**, uncheck the **Precompiled header** check box if it's checked. Check the **Empty project** box.
7. Choose **OK** to create the project.

To create a static library project in Visual Studio 2017

1. On the menu bar, choose **File** > **New** > **Project**.
2. In the **New Project** dialog box, select **Installed** > **Visual C++** > **Windows Desktop**. In the center pane, select **Windows Desktop Wizard**.

3. Specify a name for the project—for example, *MathLibrary*—in the **Name** box. Specify a name for the solution—for example, *StaticMath*—in the **Solution Name** box. Choose the **OK** button.
4. In the **Windows Desktop Project** dialog, under **Application type**, select **Static Library (.lib)**.
5. Under **Additional Options**, uncheck the **Precompiled header** check box if it's checked. Check the **Empty project** box.
6. Choose **OK** to create the project.

To create a static library project in Visual Studio 2015

1. On the menu bar, choose **File** > **New** > **Project**.
2. In the **New Project** dialog box, select **Installed** > **Templates** > **Visual C++** > **Win32**. In the center pane, select **Win32 Console Application**.
3. Specify a name for the project—for example, *MathLibrary*—in the **Name** box. Specify a name for the solution—for example, *StaticMath*—in the **Solution Name** box. Choose the **OK** button.
4. In the **Win32 Application Wizard**, choose **Next**.
5. In the **Application Settings** page, under **Application type**, select **Static library**. Under **Additional options**, uncheck the **Precompiled header** checkbox. Choose **Finish** to create the project.

Add a class to the static library

To add a class to the static library

1. To create a header file for a new class, right-click to open the shortcut menu for the **MathLibrary** project in **Solution Explorer**, and then choose **Add** > **New Item**.
2. In the **Add New Item** dialog box, select **Visual C++** > **Code**. In the center pane, select **Header File (.h)**. Specify a name for the header file—for example, *MathLibrary.h*—and then choose the **Add** button. A nearly blank header file is displayed.
3. Add a declaration for a class named `Arithmetic` to do common mathematical operations such as addition, subtraction, multiplication, and division. The code should resemble:

```
// MathLibrary.h
#pragma once

namespace MathLibrary
{
    class Arithmetic
    {
    public:
        // Returns a + b
        static double Add(double a, double b);

        // Returns a - b
        static double Subtract(double a, double b);

        // Returns a * b
        static double Multiply(double a, double b);

        // Returns a / b
        static double Divide(double a, double b);
    };
}
```

4. To create a source file for the new class, open the shortcut menu for the **MathLibrary** project in

Solution Explorer, and then choose **Add** > **New Item**.

5. In the **Add New Item** dialog box, in the center pane, select **C++ File (.cpp)**. Specify a name for the source file—for example, *MathLibrary.cpp*—and then choose the **Add** button. A blank source file is displayed.

6. Use this source file to implement the functionality for class `Arithmetic`. The code should resemble:

```
// MathLibrary.cpp
// compile with: cl /c /EHsc MathLibrary.cpp
// post-build command: lib MathLibrary.obj

#include "MathLibrary.h"

namespace MathLibrary
{
    double Arithmetic::Add(double a, double b)
    {
        return a + b;
    }

    double Arithmetic::Subtract(double a, double b)
    {
        return a - b;
    }

    double Arithmetic::Multiply(double a, double b)
    {
        return a * b;
    }

    double Arithmetic::Divide(double a, double b)
    {
        return a / b;
    }
}
```

7. To build the static library, select **Build** > **Build Solution** on the menu bar. The build creates a static library, *MathLibrary.lib*, that can be used by other programs.

> **NOTE**
>
> When you build on the Visual Studio command line, you must build the program in two steps. First, run `cl /c /EHsc MathLibrary.cpp` to compile the code and create an object file that's named *MathLibrary.obj*. (The `cl` command invokes the compiler, Cl.exe, and the `/c` option specifies compile without linking. For more information, see /c (Compile Without Linking).) Second, run `lib MathLibrary.obj` to link the code and create the static library *MathLibrary.lib*. (The `lib` command invokes the Library Manager, Lib.exe. For more information, see LIB Reference.)

Create a C++ console app that references the static library

To create a C++ console app that references the static library in Visual Studio

1. In **Solution Explorer**, right-click on the top node, **Solution 'StaticMath'**, to open the shortcut menu. Choose **Add** > **New Project** to open the **Add a New Project** dialog.

2. At the top of the dialog, set the **Project type** filter to **Console**.

3. From the filtered list of project types, choose **Console App** then choose **Next**. In the next page, enter *MathClient* in the **Name** box to specify a name for the project.

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

5. After you create a console app, an empty program is created for you. The name for the source file is the same as the name that you chose earlier. In the example, it's named `MathClient.cpp`.

To create a C++ console app that references the static library in Visual Studio 2017

1. In **Solution Explorer**, right-click on the top node, **Solution 'StaticMath'**, to open the shortcut menu. Choose **Add** > **New Project** to open the **Add a New Project** dialog box.

2. In the **Add New Project** dialog box, select **Installed** > **Visual C++** > **Windows Desktop**. In the center pane, select **Windows Desktop Wizard**.

3. Specify a name for the project—for example, *MathClient*—in the **Name** box. Choose the **OK** button.

4. In the **Windows Desktop Project** dialog, under **Application type**, select **Console Application (.exe)**.

5. Under **Additional Options**, uncheck the **Precompiled header** check box if it's checked.

6. Choose **OK** to create the project.

7. After you create a console app, an empty program is created for you. The name for the source file is the same as the name that you chose earlier. In the example, it's named `MathClient.cpp`.

To create a C++ console app that references the static library in Visual Studio 2015

1. In **Solution Explorer**, right-click on the top node, **Solution 'StaticMath'**, to open the shortcut menu. Choose **Add** > **New Project** to open the **Add a New Project** dialog box.

2. In the **Add New Project** dialog box, select **Installed** > **Visual C++** > **Win32**. In the center pane, select **Win32 Console Application**.

3. Specify a name for the project—for example, *MathClient*—in the **Name** box. Choose the **OK** button.

4. In the **Win32 Application Wizard** dialog, choose **Next**.

5. On the **Application Settings** page, under **Application type**, make sure **Console application** is selected. Under **Additional options**, uncheck **Precompiled header**, then check the **Empty Project** checkbox. Choose **Finish** to create the project.

6. To add a source file to the empty project, right-click to open the shortcut menu for the **MathClient** project in **Solution Explorer**, and then choose **Add** > **New Item**.

7. In the **Add New Item** dialog box, select **Visual C++** > **Code**. In the center pane, select **C++ File (.cpp)**. Specify a name for the source file—for example, *MathClient.cpp*—and then choose the **Add** button. A blank source file is displayed.

Use the functionality from the static library in the app

To use the functionality from the static library in the app

1. Before you can use the math routines in the static library, you must reference it. Open the shortcut menu for the **MathClient** project in **Solution Explorer**, and then choose **Add** > **Reference**.

2. The **Add Reference** dialog box lists the libraries that you can reference. The **Projects** tab lists the projects in the current solution and any libraries they reference. Open the **Projects** tab, select the **MathLibrary** check box, and then choose the **OK** button.

3. To reference the `MathLibrary.h` header file, you must modify the included directories path. In **Solution Explorer**, right-click on **MathClient** to open the shortcut menu. Choose **Properties** to open the **MathClient Property Pages** dialog box.

4. In the **MathClient Property Pages** dialog box, set the **Configuration** drop-down to **All Configurations**. Set the **Platform** drop-down to **All Platforms**.

5. Select the **Configuration Properties** > **C/C++** > **General** property page. In the **Additional Include Directories** property, specify the path of the **MathLibrary** directory, or browse for it.

   To browse for the directory path:

   a. Open the **Additional Include Directories** property value drop-down list, and then choose **Edit**.

   b. In the **Additional Include Directories** dialog box, double-click in the top of the text box. Then choose the ellipsis button (...) at the end of the line.

   c. In the **Select Directory** dialog box, navigate up a level, and then select the **MathLibrary** directory. Then choose the **Select Folder** button to save your selection.

   d. In the **Additional Include Directories** dialog box, choose the **OK** button.

   e. In the **Property Pages** dialog box, choose the **OK** button to save your changes to the project.

6. You can now use the `Arithmetic` class in this app by including the `#include "MathLibrary.h"` header in your code. Replace the contents of `MathClient.cpp` with this code:

```
// MathClient.cpp
// compile with: cl /EHsc MathClient.cpp /link MathLibrary.lib

#include <iostream>
#include "MathLibrary.h"

int main()
{
    double a = 7.4;
    int b = 99;

    std::cout << "a + b = " <<
        MathLibrary::Arithmetic::Add(a, b) << std::endl;
    std::cout << "a - b = " <<
        MathLibrary::Arithmetic::Subtract(a, b) << std::endl;
    std::cout << "a * b = " <<
        MathLibrary::Arithmetic::Multiply(a, b) << std::endl;
    std::cout << "a / b = " <<
        MathLibrary::Arithmetic::Divide(a, b) << std::endl;

    return 0;
}
```

7. To build the executable, choose **Build** > **Build Solution** on the menu bar.

Run the app

To run the app

1. Make sure that **MathClient** is selected as the default project. To select it, right-click to open the shortcut menu for **MathClient** in **Solution Explorer**, and then choose **Set as StartUp Project**.

2. To run the project, on the menu bar, choose **Debug** > **Start Without Debugging**. The output should resemble:

```
a + b = 106.4
a - b = -91.6
a * b = 732.6
a / b = 0.0747475
```

See also

Walkthrough: Creating and Using a Dynamic Link Library (C++)
Desktop Applications (Visual C++)

Walkthrough: Compile a C++/CLI program that targets the CLR in Visual Studio

9/2/2022 • 3 minutes to read • Edit Online

By using C++/CLI you can create C++ programs that use .NET classes as well as native C++ types. C++/CLI is intended for use in console applications and in DLLs that wrap native C++ code and make it accessible from .NET programs. To create a Windows user interface based on .NET, use C# or Visual Basic.

For this procedure, you can type your own C++ program or use one of the sample programs. The sample program that we use in this procedure creates a text file named textfile.txt, and saves it to the project directory.

Prerequisites

- An understanding of the fundamentals of the C++ language.
- In Visual Studio 2017 and later, C++/CLI support is an optional component. To install it, open the **Visual Studio Installer** from the Windows Start menu. Make sure that the **Desktop development with C++** tile is checked, and in the **Optional** components section, also check **C++/CLI Support**.

Create a new project

The following steps vary depending on which version of Visual Studio you are using. To see the documentation for your preferred version of Visual Studio, use the **Version** selector control. It's found at the top of the table of contents on this page.

To create a C++/CLI project in Visual Studio

1. In **Solution Explorer**, right-click on the top to open the **Create a New Project** dialog box.
2. At the top of the dialog, type **CLR** in the search box and then choose **CLR Empty Project** from the results list.
3. Choose the **Create** button to create the project.

To create a C++/CLI project in Visual Studio 2017

1. Create a new project. On the **File** menu, point to **New**, and then click **Project**.
2. From the Visual C++ project types, click **CLR**, and then click **CLR Empty Project**.
3. Type a project name. By default, the solution that contains the project has the same name as the new project, but you can enter a different name. You can enter a different location for the project if you want.
4. Click **OK** to create the new project.

To create a C++/CLI project in Visual Studio 2015

1. Create a new project. On the **File** menu, point to **New**, and then click **Project**.
2. From the Visual C++ project types, click **CLR**, and then click **CLR Empty Project**.
3. Type a project name. By default, the solution that contains the project has the same name as the new project, but you can enter a different name. You can enter a different location for the project if you want.
4. Click **OK** to create the new project.

Add a source file

1. If **Solution Explorer** isn't visible, click **Solution Explorer** on the **View** menu.
2. Add a new source file to the project:
   - Right-click the **Source Files** folder in **Solution Explorer**, point to **Add**, and click **New Item**.
   - Click **C++ File (.cpp)** and type a file name and then click **Add**.

   The **.cpp** file appears in the **Source Files** folder in **Solution Explorer** and a tabbed window appears where you type the code you want in that file.
3. Click in the newly created tab in Visual Studio and type a valid Visual C++ program, or copy and paste one of the sample programs.

   For example, you can use the How to: Write a Text File (C++/CLI) sample program (in the **File Handling and I/O** node of the Programming Guide).

   If you use the sample program, notice that you use the `gcnew` keyword instead of `new` when creating a .NET object, and that `gcnew` returns a handle (`^`) rather than a pointer (`*`):

   ```
   StreamWriter^ sw = gcnew StreamWriter(fileName);
   ```

   For more information on C++/CLI syntax, see Component Extensions for Runtime Platforms.
4. On the **Build** menu, click **Build Solution**.

   The **Output** window displays information about the compilation progress, such as the location of the build log and a message that indicates the build status.

   If you make changes and run the program without doing a build, a dialog box might indicate that the project is out of date. Select the checkbox on this dialog before you click **OK** if you want Visual Studio to always use the current versions of files instead of prompting you each time it builds the application.
5. On the **Debug** menu, click **Start without Debugging**.
6. If you used the sample program, when you run the program a command window is displayed that indicates the text file has been created.

   The **textfile.txt** text file is now located in your project directory. You can open this file by using Notepad.

   > **NOTE**
   >
   > Choosing the empty CLR project template automatically set the `/clr` compiler option. To verify this, right-click the project in **Solution Explorer** and clicking **Properties**, and then check the **Common Language Runtime support** option in the **General** node of **Configuration Properties**.

See also

C++ Language Reference
Projects and build systems

Walkthrough: Compiling a Native C++ Program on the Command Line

9/2/2022 • 10 minutes to read • Edit Online

Visual Studio includes a command-line C and C++ compiler. You can use it to create everything from basic console apps to Universal Windows Platform apps, Desktop apps, device drivers, and .NET components.

In this walkthrough, you create a basic, "Hello, World"-style C++ program by using a text editor, and then compile it on the command line. If you'd like to try the Visual Studio IDE instead of using the command line, see Walkthrough: Working with Projects and Solutions (C++) or Using the Visual Studio IDE for C++ Desktop Development.

In this walkthrough, you can use your own C++ program instead of typing the one that's shown. Or, you can use a C++ code sample from another help article.

Prerequisites

To complete this walkthrough, you must have installed either Visual Studio and the optional **Desktop development with C++** workload, or the command-line Build Tools for Visual Studio.

Visual Studio is an *integrated development environment* (IDE). It supports a full-featured editor, resource managers, debuggers, and compilers for many languages and platforms. Versions available include the free Visual Studio Community edition, and all can support C and C++ development. For information on how to download and install Visual Studio, see Install C++ support in Visual Studio.

The Build Tools for Visual Studio installs only the command-line compilers, tools, and libraries you need to build C and C++ programs. It's perfect for build labs or classroom exercises and installs relatively quickly. To install only the command-line tools, look for Build Tools for Visual Studio on the Visual Studio Downloads page.

Before you can build a C or C++ program on the command line, verify that the tools are installed, and you can access them from the command line. Visual C++ has complex requirements for the command-line environment to find the tools, headers, and libraries it uses. **You can't use Visual C++ in a plain command prompt window** without doing some preparation. Fortunately, Visual C++ installs shortcuts for you to launch a developer command prompt that has the environment set up for command line builds. Unfortunately, the names of the developer command prompt shortcuts and where they're located are different in almost every version of Visual C++ and on different versions of Windows. Your first walkthrough task is finding the right one to use.

> **NOTE**
>
> A developer command prompt shortcut automatically sets the correct paths for the compiler and tools, and for any required headers and libraries. You must set these environment values yourself if you use a regular **Command Prompt** window. For more information, see Use the MSVC toolset from the command line. We recommend you use a developer command prompt shortcut instead of building your own.

Open a developer command prompt

1. If you have installed Visual Studio 2017 or later on Windows 10 or later, open the Start menu and choose **All apps**. Scroll down and open the **Visual Studio** folder (not the Visual Studio application). Choose **Developer Command Prompt for VS** to open the command prompt window.

If you have installed Microsoft Visual C++ Build Tools 2015 on Windows 10 or later, open the **Start** menu and choose **All apps**. Scroll down and open the **Visual C++ Build Tools** folder. Choose **Visual C++ 2015 x86 Native Tools Command Prompt** to open the command prompt window.

You can also use the Windows search function to search for "developer command prompt" and choose one that matches your installed version of Visual Studio. Use the shortcut to open the command prompt window.

2. Next, verify that the Visual C++ developer command prompt is set up correctly. In the command prompt window, enter `cl` and verify that the output looks something like this:

```
C:\Program Files (x86)\Microsoft Visual Studio\2017\Enterprise>cl
Microsoft (R) C/C++ Optimizing Compiler Version 19.10.25017 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

usage: cl [ option... ] filename... [ /link linkoption... ]
```

There may be differences in the current directory or version numbers. These values depend on the version of Visual C++ and any updates installed. If the above output is similar to what you see, then you're ready to build C or C++ programs at the command line.

> **NOTE**
> If you get an error such as "'cl' is not recognized as an internal or external command, operable program or batch file," error C1034, or error LNK1104 when you run the `cl` command, then either you are not using a developer command prompt, or something is wrong with your installation of Visual C++. You must fix this issue before you can continue.

If you can't find the developer command prompt shortcut, or if you get an error message when you enter `cl`, then your Visual C++ installation may have a problem. Try reinstalling the Visual C++ component in Visual Studio, or reinstall the Microsoft Visual C++ Build Tools. Don't go on to the next section until the `cl` command works. For more information about installing and troubleshooting Visual C++, see Install Visual Studio.

> **NOTE**
> Depending on the version of Windows on the computer and the system security configuration, you might have to right-click to open the shortcut menu for the developer command prompt shortcut and then choose **Run as administrator** to successfully build and run the program that you create by following this walkthrough.

Create a Visual C++ source file and compile it on the command line

1. In the developer command prompt window, enter `md c:\hello` to create a directory, and then enter `cd c:\hello` to change to that directory. This directory is where both your source file and the compiled program get created.

2. Enter `notepad hello.cpp` in the command prompt window.

   Choose **Yes** when Notepad prompts you to create a new file. This step opens a blank Notepad window, ready for you to enter your code in a file named hello.cpp.

3. In Notepad, enter the following lines of code:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello, world, from Visual C++!" << endl;
}
```

This code is a simple program that will write one line of text on the screen and then exit. To minimize errors, copy this code and paste it into Notepad.

4. Save your work! In Notepad, on the **File** menu, choose **Save**.

   Congratulations, you've created a C++ source file, hello.cpp, that is ready to compile.

5. Switch back to the developer command prompt window. Enter `dir` at the command prompt to list the contents of the c:\hello directory. You should see the source file hello.cpp in the directory listing, which looks something like:

```
c:\hello>dir
 Volume in drive C has no label.
 Volume Serial Number is CC62-6545

 Directory of c:\hello

05/24/2016  05:36 PM    <DIR>          .
05/24/2016  05:36 PM    <DIR>          ..
05/24/2016  05:37 PM               115 hello.cpp
               1 File(s)            115 bytes
               2 Dir(s)  571,343,446,016 bytes free
```

   The dates and other details will differ on your computer.

   NOTE

   If you don't see your source code file, `hello.cpp`, make sure the current working directory in your command prompt is the `C:\hello` directory you created. Also make sure that this is the directory where you saved your source file. And make sure that you saved the source code with a `.cpp` file name extension, not a `.txt` extension. Your source file gets saved in the current directory as a `.cpp` file automatically if you open Notepad at the command prompt by using the `notepad hello.cpp` command. Notepad's behavior is different if you open it another way: By default, Notepad appends a `.txt` extension to new files when you save them. It also defaults to saving files in your *Documents* directory. To save your file with a `.cpp` extension in Notepad, choose **File** > **Save As**. In the **Save As** dialog, navigate to your `C:\hello` folder in the directory tree view control. Then use the **Save as type** dropdown control to select **All Files (*.*)**. Enter `hello.cpp` in the **File name** edit control, and then choose **Save** to save the file.

6. At the developer command prompt, enter `cl /EHsc hello.cpp` to compile your program.

   The cl.exe compiler generates an .obj file that contains the compiled code, and then runs the linker to create an executable program named hello.exe. This name appears in the lines of output information that the compiler displays. The output of the compiler should look something like:

```
c:\hello>cl /EHsc hello.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.10.25017 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

hello.cpp
Microsoft (R) Incremental Linker Version 14.10.25017.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:hello.exe
hello.obj
```

> **NOTE**
>
> If you get an error such as "'cl' is not recognized as an internal or external command, operable program or batch file," error C1034, or error LNK1104, your developer command prompt is not set up correctly. For information on how to fix this issue, go back to the **Open a developer command prompt** section.

> **NOTE**
>
> If you get a different compiler or linker error or warning, review your source code to correct any errors, then save it and run the compiler again. For information about specific errors, use the search box to look for the error number.

7. To run the hello.exe program, at the command prompt, enter `hello`.

   The program displays this text and exits:

   ```
   Hello, world, from Visual C++!
   ```

   Congratulations, you've compiled and run a C++ program by using the command-line tools.

Next steps

This "Hello, World" example is about as simple as a C++ program can get. Real world programs usually have header files, more source files, and link to libraries.

You can use the steps in this walkthrough to build your own C++ code instead of typing the sample code shown. These steps also let you build many C++ code sample programs that you find elsewhere. You can put your source code and build your apps in any writeable directory. By default, the Visual Studio IDE creates projects in your user folder, in a *source\repos* subfolder. Older versions may put projects in a *Documents\Visual Studio <version>\Projects* folder.

To compile a program that has additional source code files, enter them all on the command line, like:

```
cl /EHsc file1.cpp file2.cpp file3.cpp
```

The `/EHsc` command-line option instructs the compiler to enable standard C++ exception handling behavior. Without it, thrown exceptions can result in undestroyed objects and resource leaks. For more information, see /EH (Exception Handling Model).

When you supply additional source files, the compiler uses the first input file to create the program name. In this case, it outputs a program called file1.exe. To change the name to program1.exe, add an /out linker option:

```
cl /EHsc file1.cpp file2.cpp file3.cpp /link /out:program1.exe
```

And to catch more programming mistakes automatically, we recommend you compile by using either the /W3 or /W4 warning level option:

```
cl /W4 /EHsc file1.cpp file2.cpp file3.cpp /link /out:program1.exe
```

The compiler, cl.exe, has many more options. You can apply them to build, optimize, debug, and analyze your code. For a quick list, enter `cl /?` at the developer command prompt. You can also compile and link separately and apply linker options in more complex build scenarios. For more information on compiler and linker options and usage, see C/C++ Building Reference.

You can use NMAKE and makefiles, MSBuild and project files, or CMake, to configure and build more complex projects on the command line. For more information on using these tools, see NMAKE Reference, MSBuild, and CMake projects in Visual Studio.

The C and C++ languages are similar, but not the same. The MSVC compiler uses a simple rule to determine which language to use when it compiles your code. By default, the MSVC compiler treats files that end in `.c` as C source code, and files that end in `.cpp` as C++ source code. To force the compiler to treat all files as C++ independent of file name extension, use the /TP compiler option.

The MSVC compiler includes a C Runtime Library (CRT) that conforms to the ISO C99 standard, with minor exceptions. Portable code generally compiles and runs as expected. Certain obsolete library functions, and several POSIX function names, are deprecated by the MSVC compiler. The functions are supported, but the preferred names have changed. For more information, see Security Features in the CRT and Compiler Warning (level 3) C4996.

See also

C++ Language Reference
Projects and build systems
MSVC Compiler Options

Walkthrough: Compile a C program on the command line

9/2/2022 • 10 minutes to read • Edit Online

The Visual Studio build tools include a C compiler that you can use to create everything from basic console programs to full Windows Desktop applications, mobile apps, and more. Microsoft C/C++ (MSVC) is a C and C++ compiler that, in its latest versions, conforms to some of the latest C language standards, including C11 and C17.

This walkthrough shows how to create a basic, "Hello, World"-style C program by using a text editor, and then compile it on the command line. If you'd rather work in C++ on the command line, see Walkthrough: Compiling a Native C++ Program on the Command Line. If you'd like to try the Visual Studio IDE instead of using the command line, see Walkthrough: Working with Projects and Solutions (C++) or Using the Visual Studio IDE for C++ Desktop Development.

Prerequisites

To complete this walkthrough, you must have installed either Visual Studio or the Build Tools for Visual Studio and the optional Desktop development with C++ workload.

Visual Studio is a powerful integrated development environment that supports a full-featured editor, resource managers, debuggers, and compilers for many languages and platforms. For information on these features and how to download and install Visual Studio, including the free Visual Studio Community edition, see Install Visual Studio.

The Build Tools for Visual Studio version of Visual Studio installs only the command-line toolset, the compilers, tools, and libraries you need to build C and C++ programs. It's perfect for build labs or classroom exercises and installs relatively quickly. To install only the command-line toolset, download Build Tools for Visual Studio from the Visual Studio downloads page and run the installer. In the Visual Studio installer, select the **Desktop development with C++** workload (in older versions of Visual Studio, select the **C++ build tools** workload), and choose **Install**.

When you've installed the tools, there's another tool you'll use to build a C or C++ program on the command line. MSVC has complex requirements for the command-line environment to find the tools, headers, and libraries it uses. **You can't use MSVC in a plain command prompt window** without some preparation. You need a *developer command prompt* window, which is a regular command prompt window that has all the required environment variables set. Fortunately, Visual Studio installs shortcuts for you to launch developer command prompts that have the environment set up for command line builds. Unfortunately, the names of the developer command prompt shortcuts and where they're located are different in almost every version of Visual Studio and on different versions of Windows. Your first walkthrough task is to find the right shortcut to use.

> **NOTE**
>
> A developer command prompt shortcut automatically sets the correct paths for the compiler and tools, and for any required headers and libraries. Some of these values are different for each build configuration. You must set these environment values yourself if you don't use one of the shortcuts. For more information, see Use the MSVC toolset from the command line. Because the build environment is complex, we strongly recommend you use a developer command prompt shortcut instead of building your own.

These instructions vary depending on which version of Visual Studio you're using. To see the documentation for

your preferred version of Visual Studio, use the **Version** selector control. It's found at the top of the table of contents on this page.

Open a developer command prompt in Visual Studio 2022

If you've installed Visual Studio 2022 on Windows 10 or later, open the Start menu, and choose **All apps**. Then, scroll down and open the **Visual Studio 2022** folder (not the Visual Studio 2022 app). Choose **Developer Command Prompt for VS 2022** to open the command prompt window.

Open a developer command prompt in Visual Studio 2019

If you've installed Visual Studio 2019 on Windows 10 or later, open the Start menu, and choose **All apps**. Then, scroll down and open the **Visual Studio 2019** folder (not the Visual Studio 2019 app). Choose **Developer Command Prompt for VS 2019** to open the command prompt window.

Open a developer command prompt in Visual Studio 2017

If you've installed Visual Studio 2017 on Windows 10 or later, open the Start menu, and choose **All apps**. Then, scroll down and open the **Visual Studio 2017** folder (not the Visual Studio 2017 app). Choose **Developer Command Prompt for VS 2017** to open the command prompt window.

Open a developer command prompt in Visual Studio 2015

If you've installed Microsoft Visual C++ Build Tools 2015 on Windows 10 or later, open the Start menu, and choose **All apps**. Then, scroll down and open the **Visual C++ Build Tools** folder. Choose **Visual C++ 2015 x86 Native Tools Command Prompt** to open the command prompt window.

If you're using a different version of Windows, look in your Start menu or Start page for a Visual Studio tools folder that contains a developer command prompt shortcut. You can also use the Windows search function to search for "developer command prompt" and choose one that matches your installed version of Visual Studio. Use the shortcut to open the command prompt window.

Next, verify that the developer command prompt is set up correctly. In the command prompt window, enter `cl` (or `CL`, case doesn't matter for the compiler name, but it does matter for compiler options). The output should look something like this:

```
C:\Program Files (x86)\Microsoft Visual Studio\2017\Enterprise>cl
Microsoft (R) C/C++ Optimizing Compiler Version 19.10.25017 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

usage: cl [ option... ] filename... [ /link linkoption... ]
```

There may be differences in the current directory or version numbers, depending on the version of Visual Studio and any updates installed. If the above output is similar to what you see, then you're ready to build C or C++ programs at the command line.

> **NOTE**
> If you get an error such as "'cl' is not recognized as an internal or external command, operable program or batch file," error C1034, or error LNK1104 when you run the **cl** command, then either you are not using a developer command prompt, or something is wrong with your installation of Visual Studio. You must fix this issue before you can continue.

If you can't find the developer command prompt shortcut, or if you get an error message when you enter `cl`, then your Visual Studio installation may have a problem. If you're using Visual Studio 2017 or later, try

reinstalling the **Desktop development with C++** workload in the Visual Studio installer. For details, see Install C++ support in Visual Studio. Or, reinstall the Build Tools from the Visual Studio downloads page. Don't go on to the next section until the `cl` command works. For more information about installing and troubleshooting Visual Studio, see Install Visual Studio.

> **NOTE**
>
> Depending on the version of Windows on the computer and the system security configuration, you might have to right-click to open the shortcut menu for the developer command prompt shortcut and then choose **Run as Administrator** to successfully build and run the program that you create by following this walkthrough.

Create a C source file and compile it on the command line

1. In the developer command prompt window, enter `cd c:\` to change the current working directory to the root of your C: drive. Next, enter `md c:\hello` to create a directory, and then enter `cd c:\hello` to change to that directory. This directory will hold your source file and the compiled program.

2. Enter `notepad hello.c` at the developer command prompt. In the Notepad alert dialog that pops up, choose **Yes** to create a new *`hello.c`* file in your working directory.

3. In Notepad, enter the following lines of code:

```
#include <stdio.h>

int main()
{
    printf("Hello, World! This is a native C program compiled on the command line.\n");
    return 0;
}
```

4. On the Notepad menu bar, choose **File** > **Save** to save *`hello.c`* in your working directory.

5. Switch back to the developer command prompt window. Enter `dir` at the command prompt to list the contents of the *`c:\hello`* directory. You should see the source file *`hello.c`* in the directory listing, which looks something like:

```
C:\hello>dir
 Volume in drive C has no label.
 Volume Serial Number is CC62-6545

 Directory of C:\hello

10/02/2017  03:46 PM    <DIR>          .
10/02/2017  03:46 PM    <DIR>          ..
10/02/2017  03:36 PM               143 hello.c
               1 File(s)            143 bytes
               2 Dir(s)  514,900,566,016 bytes free
```

   The dates and other details will differ on your computer. If you don't see your source code file, *`hello.c`*, make sure you've changed to the *`c:\hello`* directory you created, and in Notepad, make sure that you saved your source file in this directory. Also make sure that you saved the source code with a *`.c`* file name extension, not a *`.txt`* extension.

6. To compile your program, enter `cl hello.c` at the developer command prompt.

   You can see the executable program name, hello.exe, in the lines of output information that the compiler

displays:

```
c:\hello>cl hello.c
Microsoft (R) C/C++ Optimizing Compiler Version 19.10.25017 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

hello.c
Microsoft (R) Incremental Linker Version 14.10.25017.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:hello.exe
hello.obj
```

> **NOTE**
>
> If you get an error such as "'cl' is not recognized as an internal or external command, operable program or batch file," error C1034, or error LNK1104, your developer command prompt is not set up correctly. For information on how to fix this issue, go back to the **Open a developer command prompt** section.
>
> If you get a different compiler or linker error or warning, review your source code to correct any errors, then save it and run the compiler again. For information about specific errors, use the search box at the top of this page to look for the error number.

7. To run your program, enter `hello` at the command prompt.

   The program displays this text and then exits:

   ```
   Hello, World! This is a native C program compiled on the command line.
   ```

   Congratulations, you've compiled and run a C program by using the command line.

Next steps

This "Hello, World" example is about as basic as a C program can get. Real world programs have header files and more source files, link in libraries, and do useful work.

You can use the steps in this walkthrough to build your own C code instead of typing the sample code shown. You can also build many C code sample programs that you find elsewhere. To compile a program that has more source code files, enter them all on the command line:

```
cl file1.c file2.c file3.c
```

The compiler outputs a program called *`file1.exe`*. To change the name to *`program1.exe`*, add an /out linker option:

```
cl file1.c file2.c file3.c /link /out:program1.exe
```

And to catch more programming mistakes automatically, we recommend you compile by using either the /W3 or /W4 warning level option:

```
cl /W4 file1.c file2.c file3.c /link /out:program1.exe
```

The compiler, cl.exe, has many more options you can apply to build, optimize, debug, and analyze your code. For a quick list, enter `cl /?` at the developer command prompt. You can also compile and link separately and apply linker options in more complex build scenarios. For more information on compiler and linker options and usage, see C/C++ Building Reference.

You can use NMAKE and makefiles, or MSBuild and project files to configure and build more complex projects

on the command line. For more information on using these tools, see NMAKE Reference and MSBuild.

The C and C++ languages are similar, but not the same. The Microsoft C/C++ compiler (MSVC) uses a basic rule to determine which language to use when it compiles your code. By default, the MSVC compiler treats all files that end in `.c` as C source code, and all files that end in `.cpp` as C++ source code. To force the compiler to treat all files as C no matter the file name extension, use the /TC compiler option.

By default, MSVC is compatible with the ANSI C89 and ISO C99 standards, but not strictly conforming. In most cases, portable C code will compile and run as expected. The compiler provides optional support for the changes in ISO C11/C17. To compile with C11/C17 support, use the compiler flag `/std:c11` or `/std:c17`. C11/C17 support requires Windows SDK 10.0.20201.0 or later. Windows SDK 10.0.22000.0 or later is recommended. You can download the latest SDK from the Windows SDK page. For more information, and instructions on how to install and use this SDK for C development, see Install C11 and C17 support in Visual Studio.

Certain library functions and POSIX function names are deprecated by MSVC. The functions are supported, but the preferred names have changed. For more information, see Security Features in the CRT and Compiler Warning (level 3) C4996.

See also

Walkthrough: Creating a Standard C++ Program (C++)
C Language Reference
Projects and build systems
Compatibility

Walkthrough: Compiling a C++/CX Program on the Command Line

9/2/2022 • 2 minutes to read • Edit Online

> **NOTE**
> For new UWP apps and components, we recommend that you use C++/WinRT, a standard C++17 language projection for Windows Runtime APIs. C++/WinRT is available in the Windows SDK from version 1803 (10.0.17134.0) onward. C++/WinRT is implemented entirely in header files, and is designed to provide you with first-class access to the modern Windows API.

The Microsoft C++ compiler (MSVC) supports C++ component extensions (C++/CX), which has additional types and operators to target the Windows Runtime programming model. You can use C++/CX to build apps for Universal Windows Platform (UWP), and Windows desktop. For more information, see A Tour of C++/CX and Component Extensions for Runtime Platforms.

In this walkthrough, you use a text editor to create a basic C++/CX program, and then compile it on the command line. (You can use your own C++/CX program instead of typing the one that's shown, or you can use a C++/CX code sample from another help article. This technique is useful for building and testing small modules that have no UI elements.)

> **NOTE**
> You can also use the Visual Studio IDE to compile C++/CX programs. Because the IDE includes design, debugging, emulation, and deployment support that isn't available on the command line, we recommend that you use the IDE to build Universal Windows Platform (UWP) apps. For more information, see Create a UWP app in C++.

Prerequisites

You understand the fundamentals of the C++ language.

Compiling a C++/CX Program

To enable compilation for C++/CX, you must use the /ZW compiler option. The MSVC compiler generates an .exe file that targets the Windows Runtime, and links to the required libraries.

To compile a C++/CX application on the command line

1. Open a **Developer Command Prompt** window. (On the **Start** window, open **Apps**. Open the **Visual Studio Tools** folder under your version of Visual Studio, and then choose the **Developer Command Prompt** shortcut.) For more information about how to open a Developer Command Prompt window, see Use the MSVC toolset from the command line.

   Administrator credentials may be required to successfully compile the code, depending on the computer's operating system and configuration. To run the Command Prompt window as an administrator, open the shortcut menu for **Developer Command Prompt** and then choose **Run as administrator**.

2. At the command prompt, enter **notepad basiccx.cpp**.

   Choose **Yes** when you're prompted to create a file.

3. In Notepad, enter these lines:

```
using namespace Platform;

int main(Platform::Array<Platform::String^>^ args)
{
    Platform::Details::Console::WriteLine("This is a C++/CX program.");
}
```

4. On the menu bar, choose **File** > **Save**.

   You've created a C++ source file that uses the Windows Runtime Platform namespace namespace.

5. At the command prompt, enter **cl /EHsc /ZW basiccx.cpp /link /SUBSYSTEM:CONSOLE**. The cl.exe compiler compiles the source code into an .obj file, and then runs the linker to generate an executable program named basiccx.exe. (The /EHsc compiler option specifies the C++ exception-handling model, and the /link flag specifies a console application.)

6. To run the basiccx.exe program, at the command prompt, enter **basiccx**.

   The program displays this text and exits:

```
This is a C++/CX program.
```

See also

Projects and build systems
MSVC Compiler Options

Walkthrough: Compiling a C++/CLI Program on the Command Line

9/2/2022 • 2 minutes to read • Edit Online

You can create Visual C++ programs that target the Common Language Runtime (CLR) and use the .NET Framework, and build them on the command line. Visual C++ supports the C++/CLI programming language, which has additional types and operators to target the .NET programming model. For general information about the C++/CLI language, see .NET Programming with C++/CLI (Visual C++).

In this walkthrough, you use a text editor to create a basic C++/CLI program, and then compile it on the command line. (You can use your own C++/CLI program instead of typing the one that's shown, or you can use a C++/CLI code sample from another help article. This technique is useful for building and testing small modules that have no UI elements.)

Prerequisites

You understand the fundamentals of the C++ language.

Compiling a C++/CLI Program

The following steps show how to compile a C++/CLI console application that uses .NET Framework classes.

To enable compilation for C++/CLI, you must use the /clr compiler option. The MSVC compiler generates an .exe file that contains MSIL code—or mixed MSIL and native code—and links to the required .NET Framework libraries.

To compile a C++/CLI application on the command line

1. Open a **Developer Command Prompt** window. For specific instructions, see To open a developer command prompt window.

   Administrator credentials may be required to successfully compile the code, depending on the computer's operating system and configuration. To run the command prompt window as an administrator, right-click to open the shortcut menu for the command prompt and then choose **More** > **Run as administrator**.

2. At the command prompt, enter `notepad basicclr.cpp`.

   Choose **Yes** when you're prompted to create a file.

3. In Notepad, enter these lines:

   ```
   int main()
   {
       System::Console::WriteLine("This is a C++/CLI program.");
   }
   ```

4. On the menu bar, choose **File** > **Save**.

   You've created a Visual C++ source file that uses a .NET Framework class (Console) in the System namespace.

5. At the command prompt, enter `cl /clr basicclr.cpp`. The cl.exe compiler compiles the source code into

an .obj file that contains MSIL, and then runs the linker to generate an executable program named basicclr.exe.

6. To run the basicclr.exe program, at the command prompt, enter `basicclr`.

   The program displays this text and exits:

```
This is a C++/CLI program.
```

See also

C++ Language Reference
Projects and build systems
MSVC Compiler Options