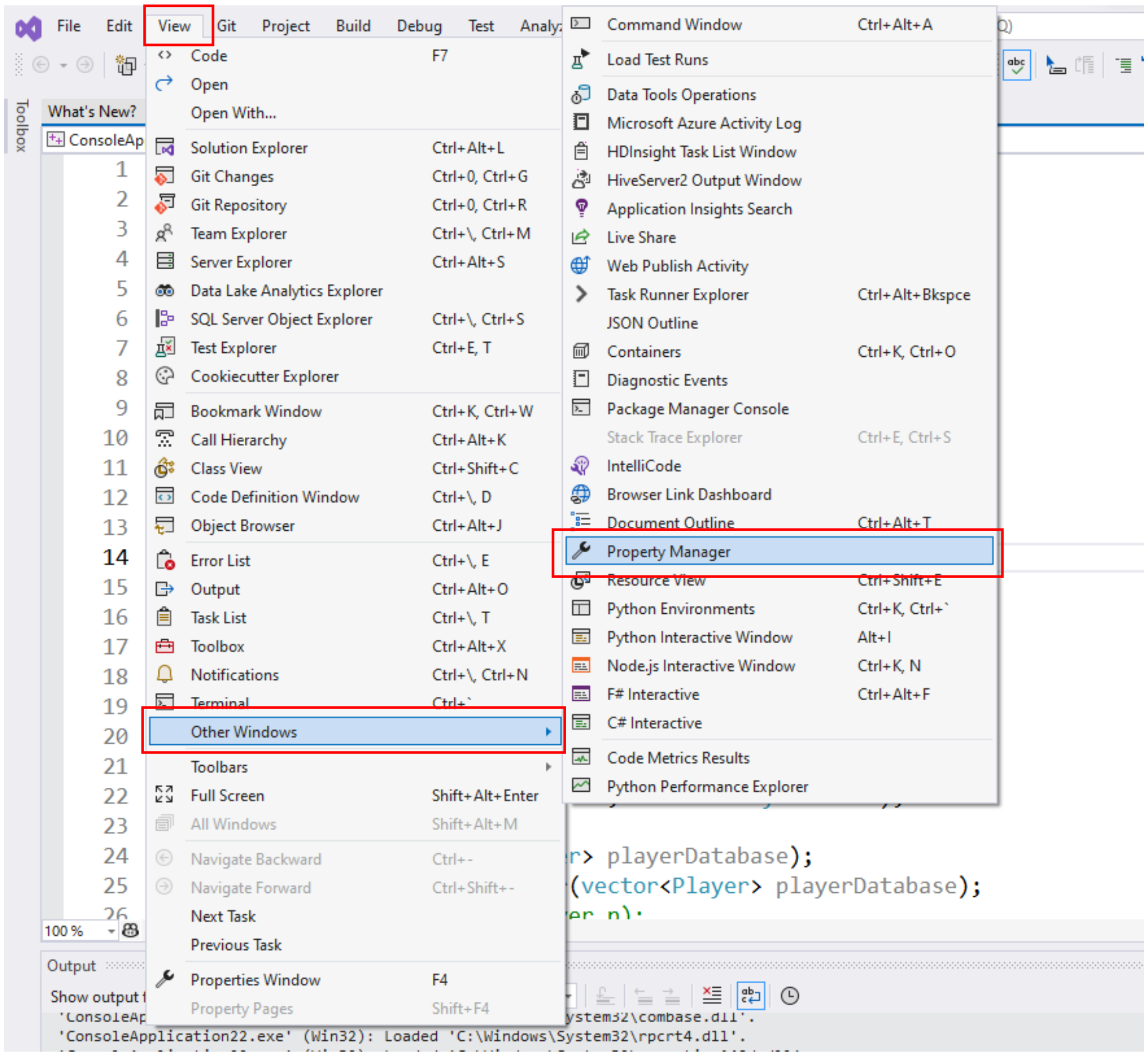
Using the C++ Boost Library inside Visual Studio

Preliminary Steps – Installing Boost, Adding a Property Sheet to a C++ Project

1. Visit the Boost website https://www.boost.org/users/download/
   Download the most current version of boost and install it in your machine (all you need to do is to unzip the file and place the folder is a designated area). In our example it was deployed in the **c:\Boost** folder as **c:\Boost\boost_1_69_0\**

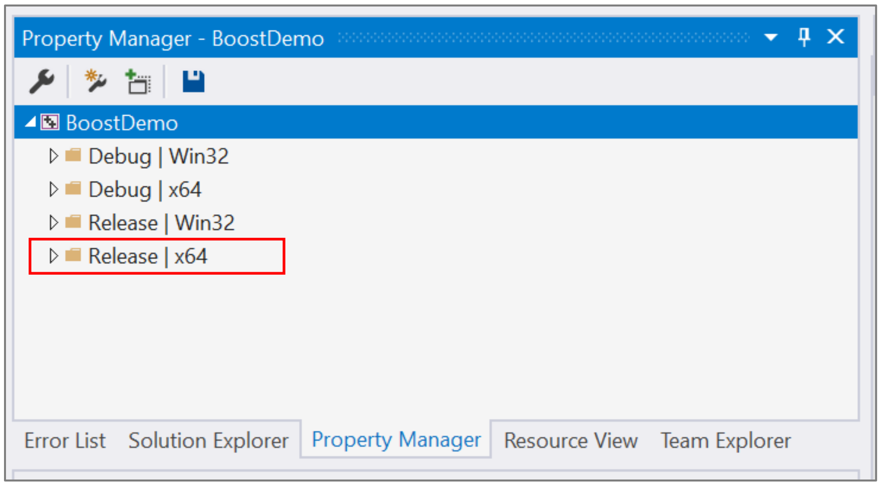
2. Create a VS C++ project. Call it **BoostDemo**.

3. Click on the *Property Manager* tab. If it is not already available on the UI, click on the menu **View > Other Screens > Property Manager**.

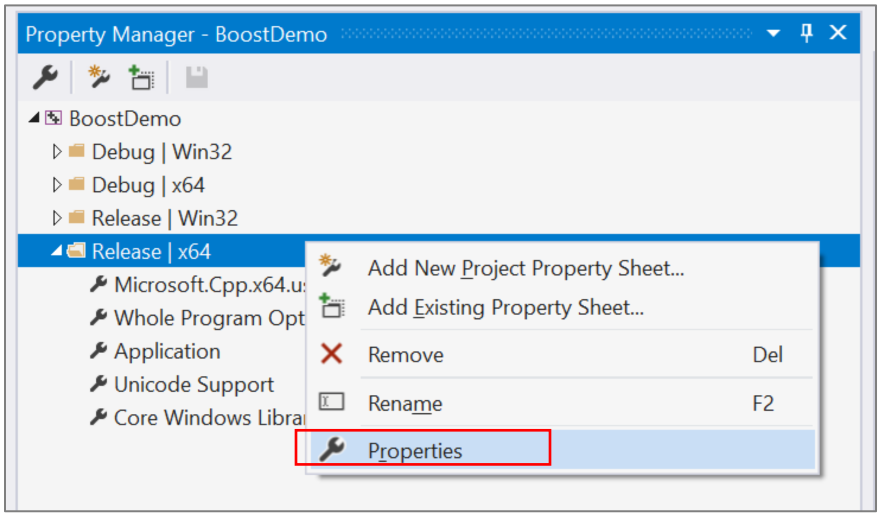

File
Edit
View
Git
Project
Build
Debug
Test
Analyz
Code F7
Open
Open With...
Solution Explorer Ctrl+Alt+L
Git Changes Ctrl+0, Ctrl+G
Git Repository Ctrl+0, Ctrl+R
Team Explorer Ctrl+\, Ctrl+M
Server Explorer Ctrl+Alt+S
Data Lake Analytics Explorer
SQL Server Object Explorer Ctrl+\, Ctrl+S
Test Explorer Ctrl+E, T
Cookiecutter Explorer
Bookmark Window Ctrl+K, Ctrl+W
Call Hierarchy Ctrl+Alt+K
Class View Ctrl+Shift+C
Code Definition Window Ctrl+\, D
Object Browser Ctrl+Alt+J
Error List Ctrl+\, E
Output Ctrl+Alt+O
Task List Ctrl+\, T
Toolbox Ctrl+Alt+X
Notifications Ctrl+\, Ctrl+N
Terminal Ctrl+`
Other Windows
Toolbars
Full Screen Shift+Alt+Enter
All Windows Shift+Alt+M
Navigate Backward Ctrl+-
Navigate Forward Ctrl+Shift+-
Next Task
Previous Task
Properties Window F4
Property Pages Shift+F4
Command Window Ctrl+Alt+A
Load Test Runs
Data Tools Operations
Microsoft Azure Activity Log
HDInsight Task List Window
HiveServer2 Output Window
Application Insights Search
Live Share
Web Publish Activity
Task Runner Explorer Ctrl+Alt+Bkspce
JSON Outline
Containers Ctrl+K, Ctrl+O
Diagnostic Events
Package Manager Console
Stack Trace Explorer Ctrl+E, Ctrl+S
IntelliCode
Browser Link Dashboard
Document Outline Ctrl+Alt+T
Property Manager
Resource View Ctrl+Shift+E
Python Environments Ctrl+K, Ctrl+`
Python Interactive Window Alt+I
Node.js Interactive Window Ctrl+K, N
F# Interactive Ctrl+Alt+F
C# Interactive
Code Metrics Results
Python Performance Explorer
Toolbox
What's New?
ConsoleAp
100 %
Output
Show output f

```
r> playerDatabase);
(vector<Player> playerDatabase);
er n);
```

```
'ConsoleAp                                ystem32\combase.dll'.
'ConsoleApplication22.exe' (Win32): Loaded 'C:\Windows\System32\rpcrt4.dll'.
```

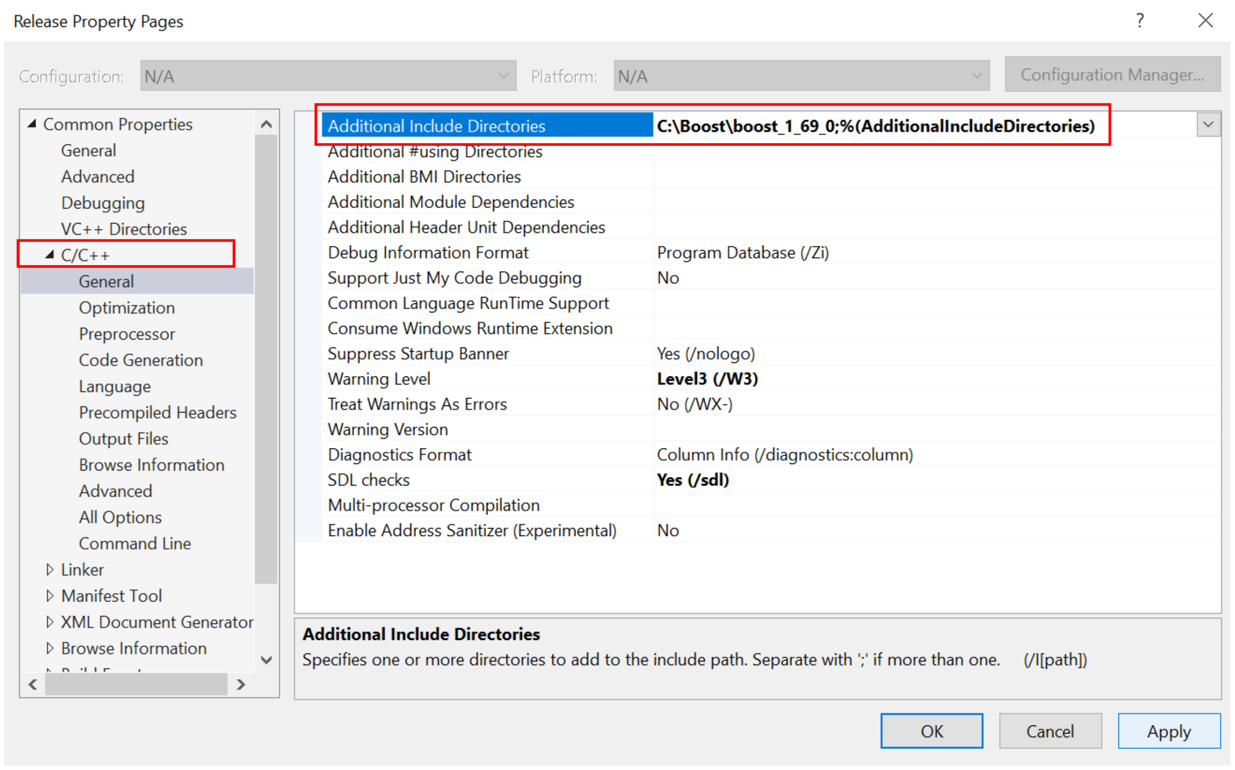


4. We have chosen to modify an existing Property Sheet (Expand the entry **Release | x64**).
Right-click on the entry **Release | x64**. From the menu, select the option **Properties**. Click it.



5. On the new page expand the **C/C++** entry. Modify the field **Additional Include Directories**.
Enter the location of your Boost library. In this example **c:\Boost\boost_1_69_0\**
Click **Apply** button.

[Alternatively, you may use the UI interface to locate and select the folder's path. Click on the *Drop-down arrow, <Edit…>, Plus-Folder icon*, pick location from *File Chooser*, click *the Select Folder* button, click *OK*.]



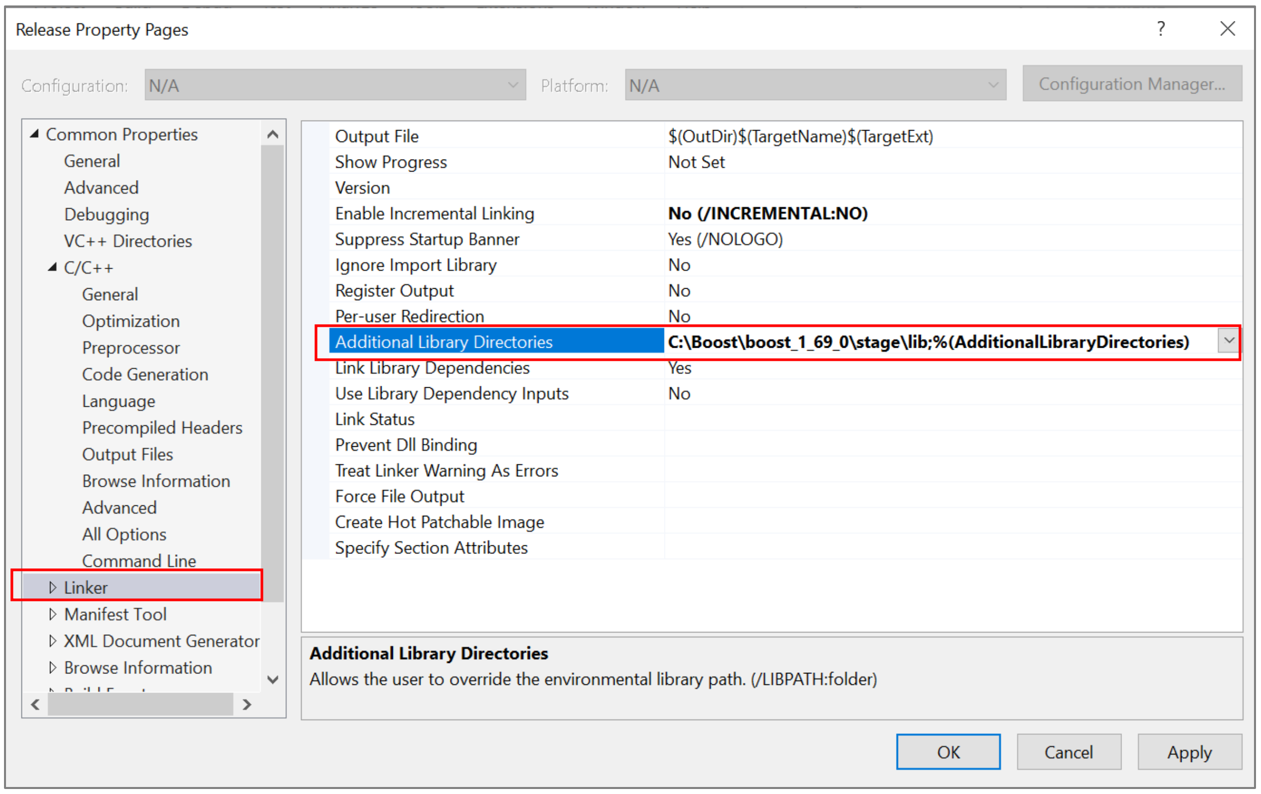
Release Property Pages

Configuration: N/A   Platform: N/A

Additional Include Directories	**C:\Boost\boost_1_69_0;%(AdditionalIncludeDirectories)**
Additional #using Directories	
Additional BMI Directories	
Additional Module Dependencies	
Additional Header Unit Dependencies	
Debug Information Format	Program Database (/Zi)
Support Just My Code Debugging	No
Common Language RunTime Support	
Consume Windows Runtime Extension	
Suppress Startup Banner	Yes (/nologo)
Warning Level	**Level3 (/W3)**
Treat Warnings As Errors	No (/WX-)
Warning Version	
Diagnostics Format	Column Info (/diagnostics:column)
SDL checks	**Yes (/sdl)**
Multi-processor Compilation	
Enable Address Sanitizer (Experimental)	No

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

6. On the above Property Page click on the **Linker** option.
   Modify the entry **Additional Library Directories**. Enter the path **c:\Boost\boost_1_69_0\stage\lib\**



Output File	$(OutDir)$(TargetName)$(TargetExt)
Show Progress	Not Set
Version	
Enable Incremental Linking	**No (/INCREMENTAL:NO)**
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	No
Additional Library Directories	**C:\Boost\boost_1_69_0\stage\lib;%(AdditionalLibraryDirectories)**
Link Library Dependencies	Yes
Use Library Dependency Inputs	No
Link Status	
Prevent Dll Binding	
Treat Linker Warning As Errors	
Force File Output	
Create Hot Patchable Image	
Specify Section Attributes	

Additional Library Directories
Allows the user to override the environmental library path. (/LIBPATH:folder)

7. Click the **OK** button. You are done!

Testing the library – Working with BIG numbers

The following app uses the Boost's Multiprecision library to work with large numbers. The app will

(1) calculate the factorial of a large number (say 40), and

(2) calculate the value of Euler's number *e* ($e = \sum_{n=0}^{\infty} \left(\frac{1}{n!}\right)$)

```
#include <boost/multiprecision/cpp_int.hpp>
#include <boost/multiprecision/cpp_dec_float.hpp>
#include <boost/math/constants/constants.hpp>

using namespace boost::multiprecision;
using namespace std;
//---------------------------------------------------------------------------------
cpp_dec_float_50 inverseFact(cpp_dec_float_50 n)
{
    cpp_dec_float_50 f = 1.0;
    for (cpp_dec_float_50 i = 1; i <= n; i++) {
        f *= i;
    }
    return 1 / f;
}
//---------------------------------------------------------------------------------
cpp_int boostFactorial(cpp_int num)
```

```cpp
{
    cpp_int fact = 1;
    for (cpp_int i = num; i > 1; --i)
        fact *= i;
    return fact;
}
//------------------------------------------------------------------------------------
int main()
{
    //calculate 40! (It will fail with a C++ long long type)
    //working with large integers
    cpp_int num = 40;
    cpp_int result = boostFactorial(num);

    cout << "\nFactorial of " << num << " is \n" << result << endl;

    //working with large precision decimal numbers
    cpp_dec_float_50 inverse = cpp_dec_float_50(1.0) / cpp_dec_float_50(result);

    cout << setprecision(numeric_limits<cpp_dec_float_50>::digits10);
    cout << "\nInverse of factorial(" << num << ") is \n" << inverse << endl;

    //Calculating an approximation of Euler's number
    cout << "\nEuler's Number\n";
    cpp_dec_float_50 sum = 0.0;
    for (cpp_dec_float_50 i = 0; i < 40; i++)
    {
        sum += inverseFact(i);
    }
    cout << "e = Sum(1/1!..1/40!) is  \n" << sum << endl;
}
```

Output

```
Factorial of 40 is

815915283247897734345611269596115894272000000000


Inverse of factorial(40) is

1.2256174391283858494235399849397569237255619644783e-48


Euler's Number

e = Sum(1/1!..1/40!) is

2.7182818284590452353602874713526624977572470936987
```

APPENDIX

Extracted from https://www.boost.org/doc/libs/1_72_0/libs/multiprecision/doc/html/boost_multiprecision/intro.html

Boost– Multiprecision Library

- The Multiprecision Library provides **integer, rational, floating-point**, and **complex** types in C++ that have more range and precision than C++'s ordinary built-in types.
- The Multiprecision types can interoperate with any fundamental (built-in) type in C++ using defined conversion rules.
- Depending on the selected type, precision may be arbitrarily large or fixed (for example, 50 or 100 decimal digits).
- There are several providers (called backends) such as GMP, MPFR, MPIR, MPC, and TomMath. These suppliers offer different capabilities. We will use plain Boost libraries.

Basic Data Types

The fundamental number types are

Type	Used for	Header
`cpp_int`	Multiprecision integers	`boost/multiprecision/cpp_int.hpp`
`cpp_rational`	Rational types	`boost/multiprecision/cpp_int.hpp`
`cpp_bin_float`	Floating-point types	`boost/multiprecision/cpp_bin_float.hpp`
`cpp_dec_float`	Floating-point types	`boost/multiprecision/cpp_dec_float.hpp`
`cpp_complex`	Complex types	`boost/multiprecision/cpp_complex.hpp`

For example, if you want a signed, 128-bit fixed size integer:

```
#include <boost/multiprecision/cpp_int.hpp>  //  Integer types.

boost::multiprecision::int128_t  my_128_bit_int;
```

Some programmers use **`typedef`** to create more convenient shorter names. For instance, you may use the following clause

```
typedef boost::multiprecision::int128_t  bigintT;

bigintT myIntVar;

myIntVar = 12345678901234567890;

cout << (myIntVar + 1) << endl;
```

Another very common coding style that reduces typing is shown below

```cpp
namespace mp = boost::multiprecision;     // Reduces typing

mp::int128_t a(3), b(4);        // 128-bit integers
mp::int512_t c(50), d;          // 512-bit integers

b = 25;

d = c * a;                      // OK, result is an int512_t

cout << (2 * d + 1) << endl;    // expression uses common arithmetical ops and big integers
```

The following code fragment illustrates how to obtain the max value that different multiprecision cpp_int types have.

```cpp
cpp_int limit1 = numeric_limits<int128_t>::max();
cout << "LIMIT128 \n" << limit1 << endl;
cout << "Has " << limit1.str().size() << " digits\n";

cpp_int limit2 = numeric_limits<int256_t>::max();
cout << "LIMIT256 \n" << limit2 << endl;
cout << "Has " << limit2.str().size() << " digits\n";

cpp_int limit3 = numeric_limits<int512_t>::max();
cout << "LIMIT512 \n" << limit3 << endl;
cout << "Has " << limit3.str().size() << " digits\n";

cpp_int limit4 = numeric_limits<int1024_t>::max();
cout << "LIMIT1024 \n" << limit4 << endl;
cout << "Has " << limit4.str().size() << " digits\n";
```

```
LIMIT int128_t
340282366920938463463374607431768211455
Has 39 digits

LIMIT int256_t
115792089237316195423570985008687907853269984665640564039457584007913129639935
Has 78 digits

LIMIT int512_t
1340780792994259709957402499820584612747936582059239337772356144372176403007354697680187429816690
342769003185818648605085375388281194656994643364900608409５
Has 155 digits

LIMIT int1024_t
1797693134862315907729305190789024733617976978942306572734300811577326758055009631327084773224075
3602112011387987139335765878976881441662249284743063947412437776789342486548527630221960124609411
9453082952085005768838150682342462881473913110540827237163350510684586298239947245938479716304835
356329624224137215
Has 309 digits
```