

# A Crash Course in Microsoft Visual C++

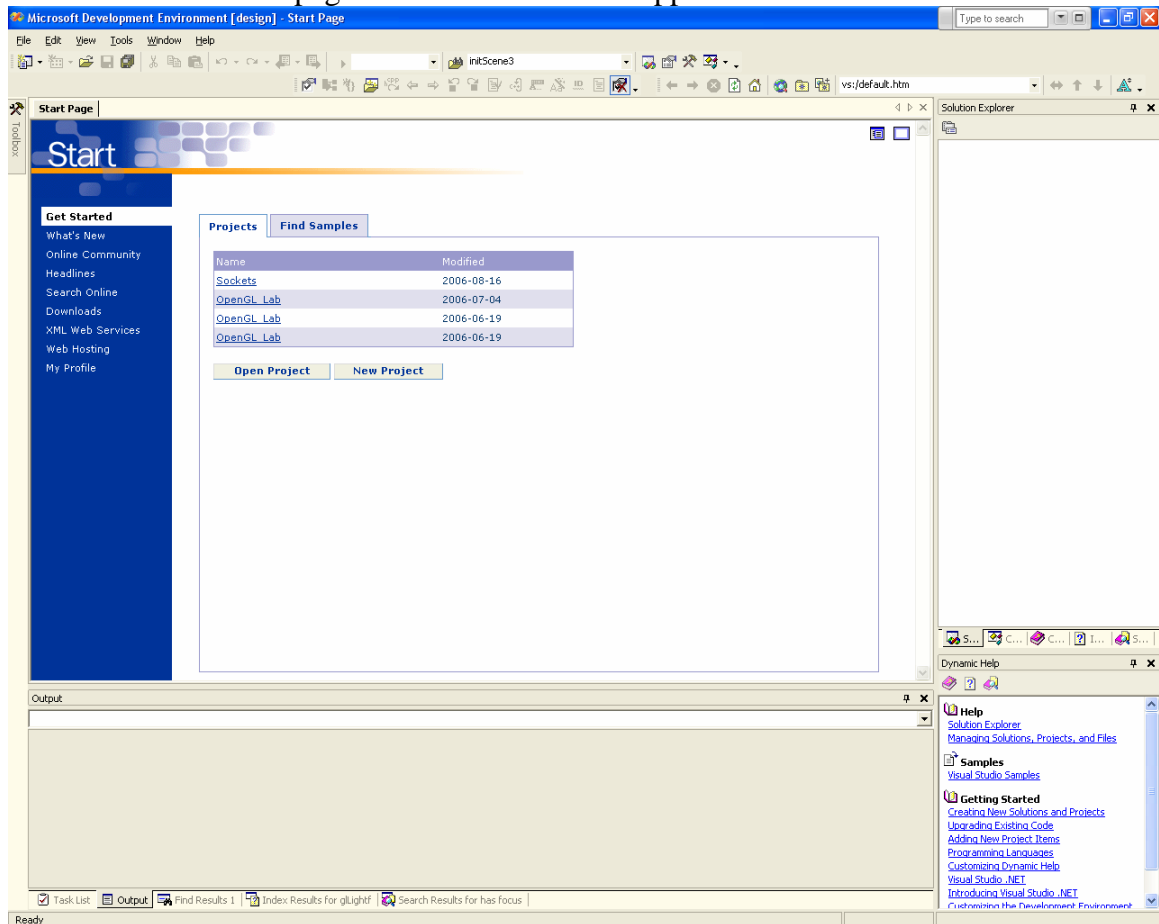
This document presents the most essential things to know in order to program with VC++. This document is intended to use with the provided “crash course in VC++.zip-file” and is part of the TDA 360 - Computer Graphics course at Chalmers.

Copying and modification of this material is free.

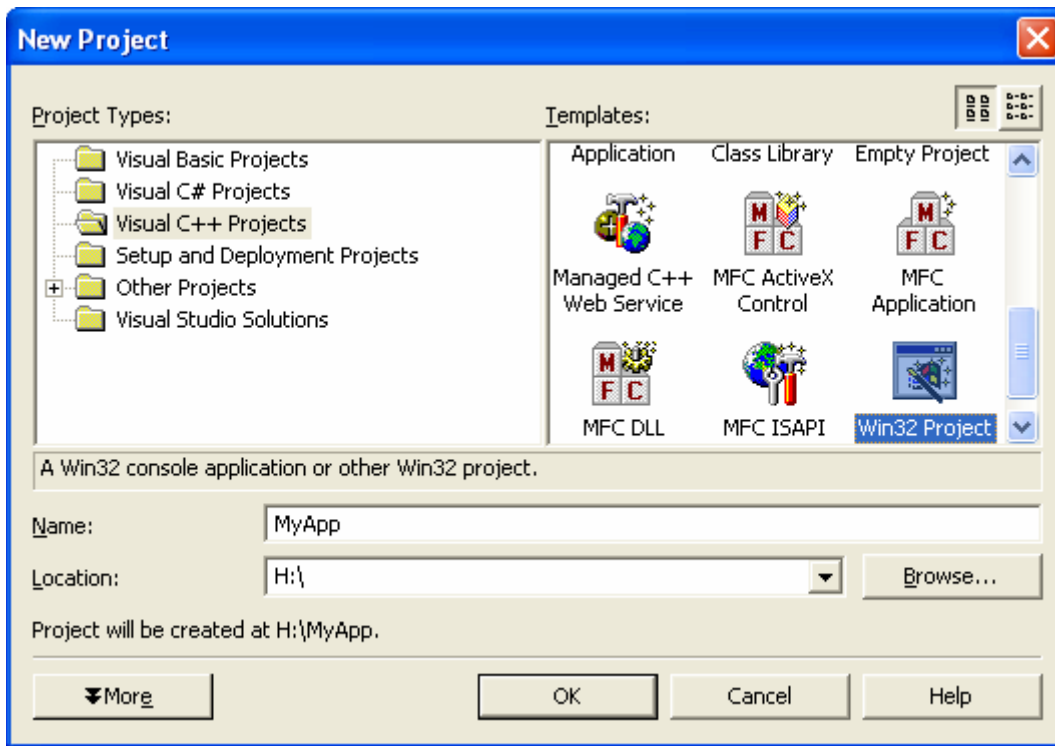
If you already have a .sln or .dsw file, skip this initial process. Otherwise, simply double click the .sln file to start VC++ or double click the .dsw file and allow VC++ to automatically convert the project.

Start Visual Studio .NET. You will find it under Start-> Programs->Programming->Microsoft Visual Studio .NET 2003...

After a few seconds a page similar to this should appear:



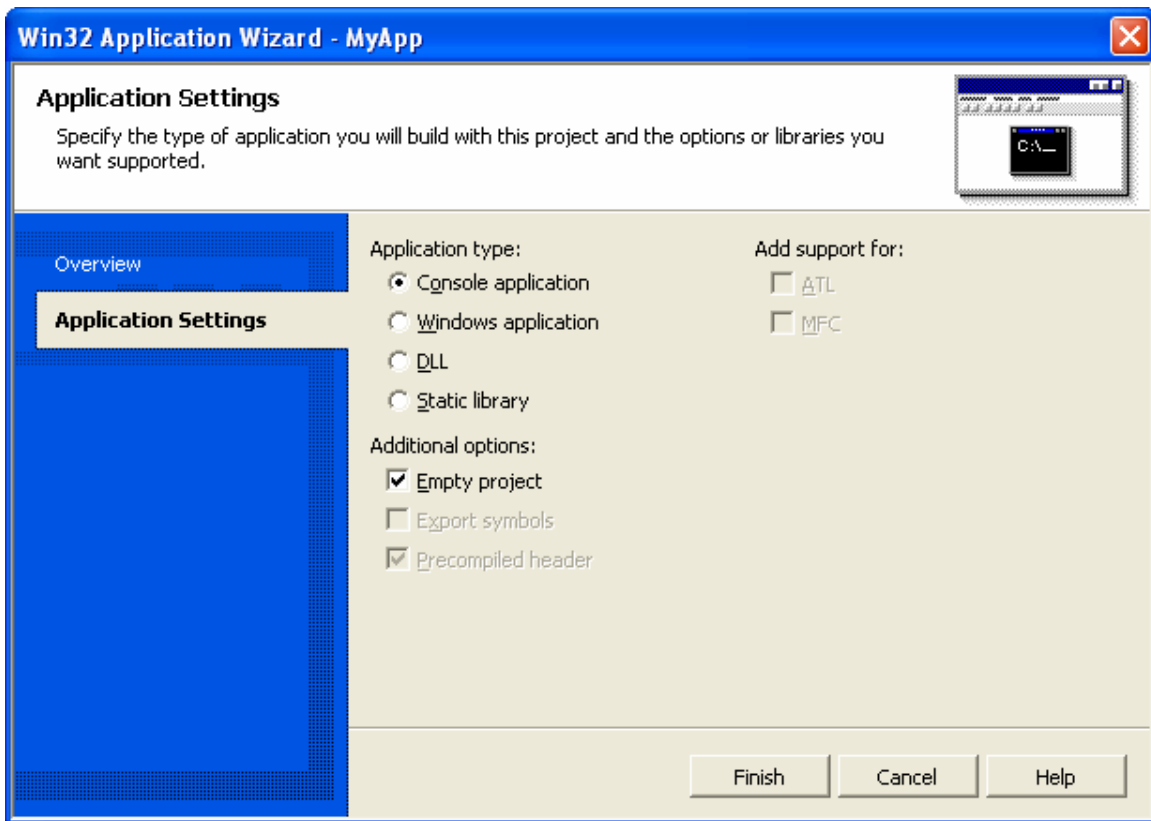
Now, create a project. Choose menu File->New->Project... and the following dialog should appear:



Be sure to select “Win32 Project”. Type the name of your project (MyApp) and the location (H:\). VC++ will create a folder H:\MyApp as the location for your project.

---

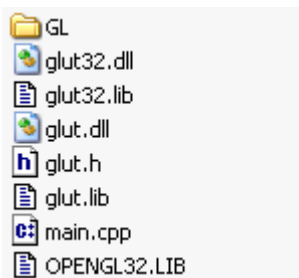
Then, select “Console application” and “Empty Project”. Otherwise we will get a lot of auto-generated code for Windows-applications, which we currently are not interested in.



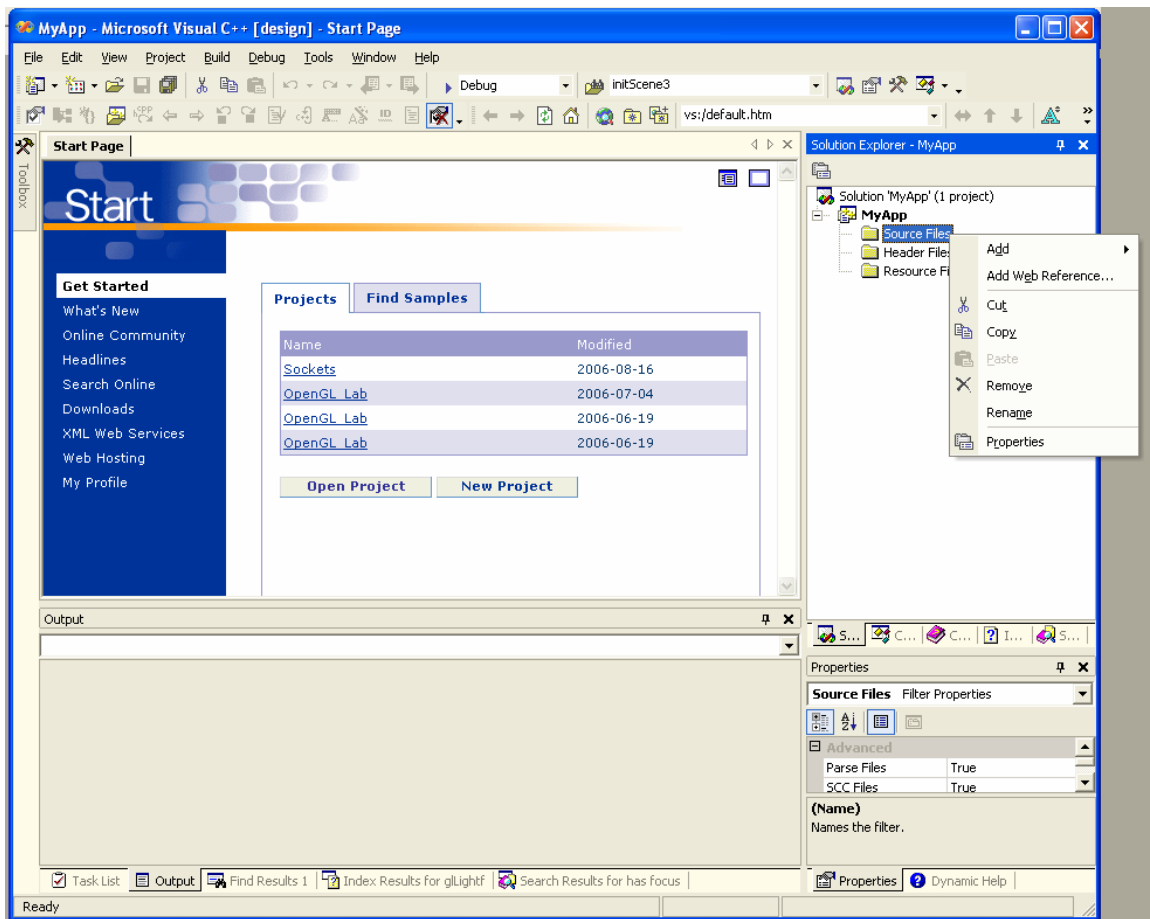
---

Copy your files to the created folder

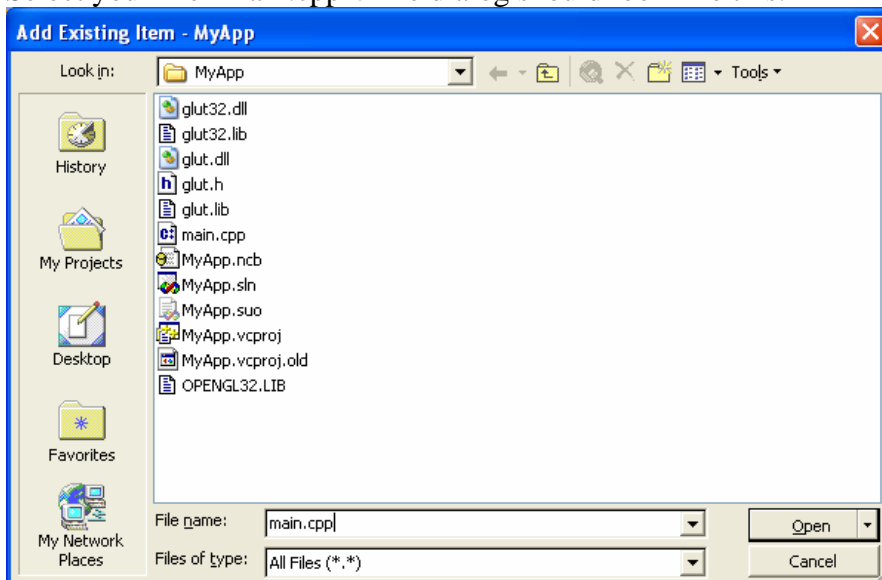
E.g. copy these files, including the directory GL, to H:\MyApp\



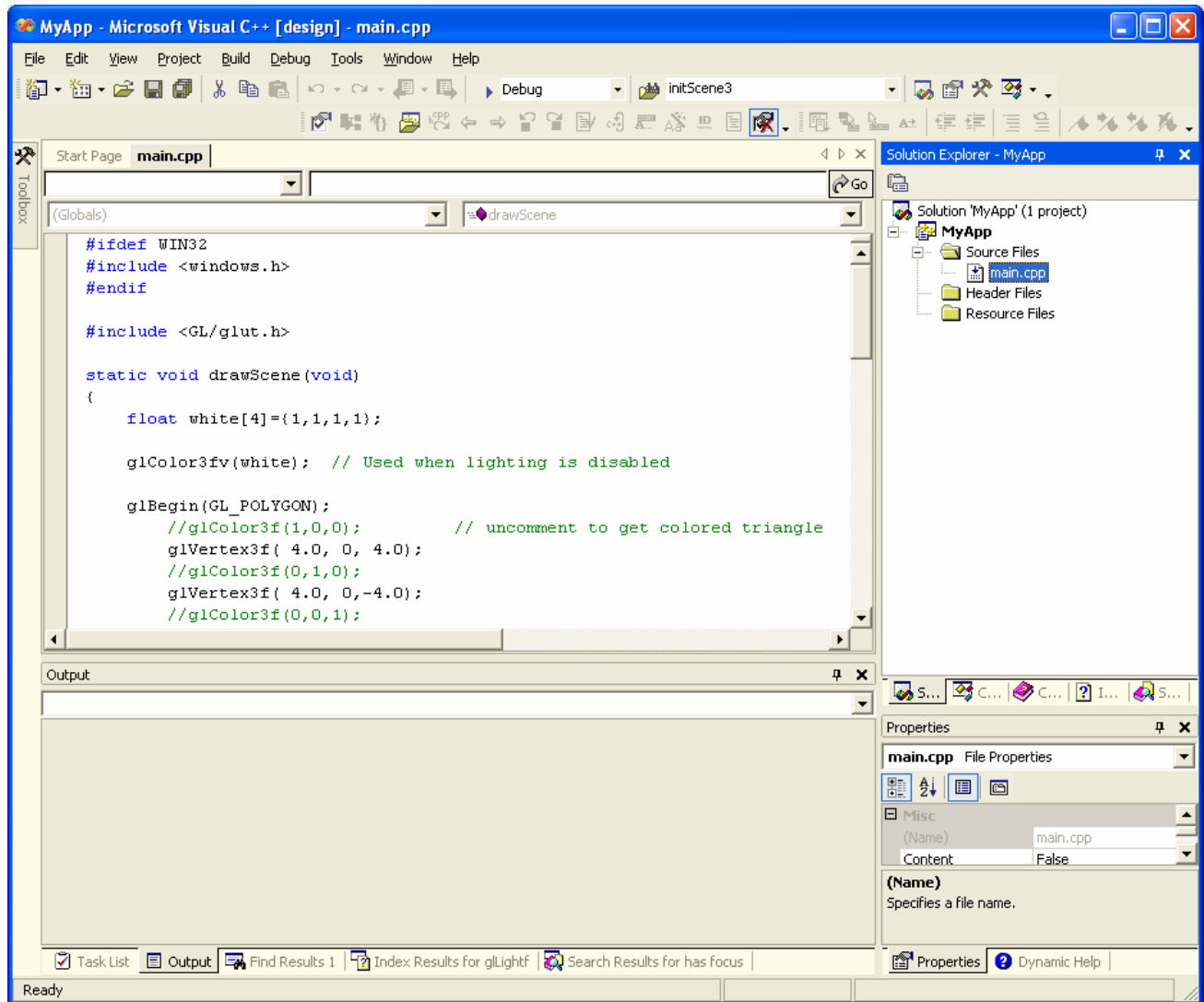
Add your cpp-file(s) to the project by right clicking on “source” and select “Add”->Add existing item...”



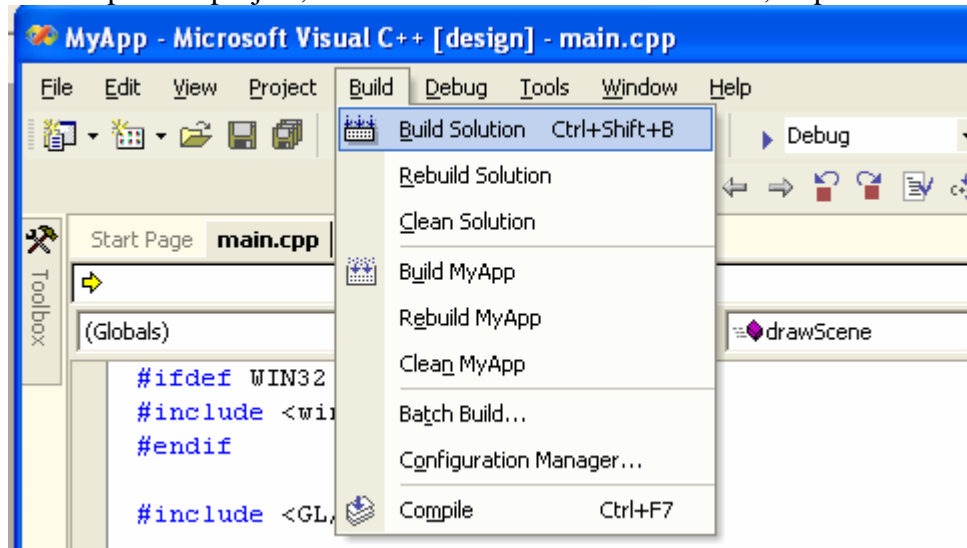
Select your file “main.cpp”. The dialog should look like this.



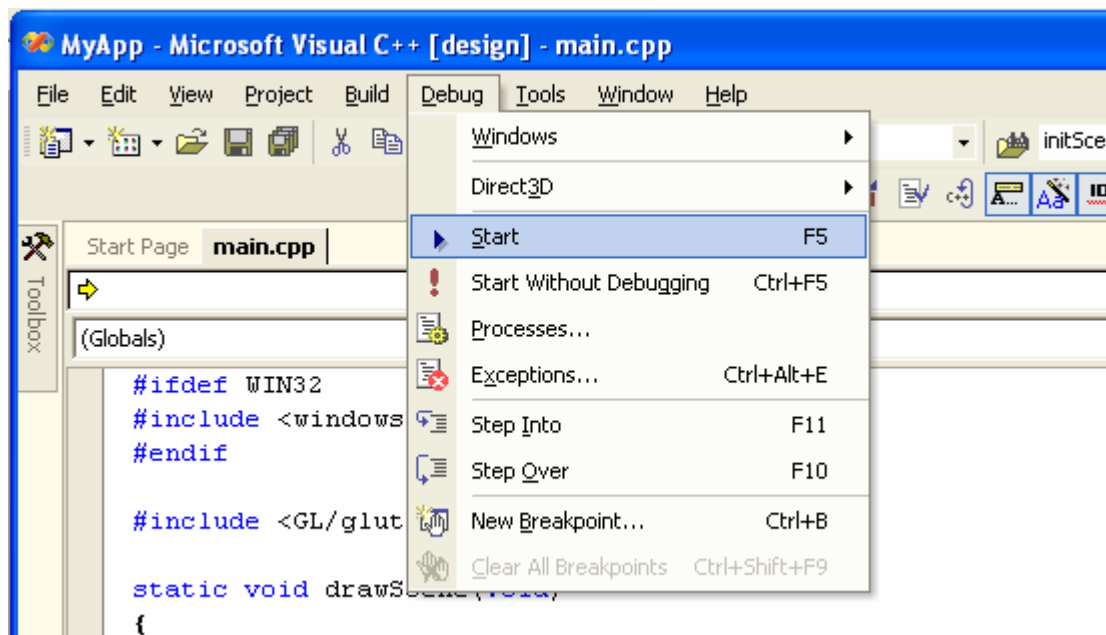
Double click on “main.cpp” under source files and your source code should appear in the editor like this:



To compile the project, select menu Build->Build Solution, or press ctrl + shift + B:



To run the application, press F5 or select menu Debug->Start.

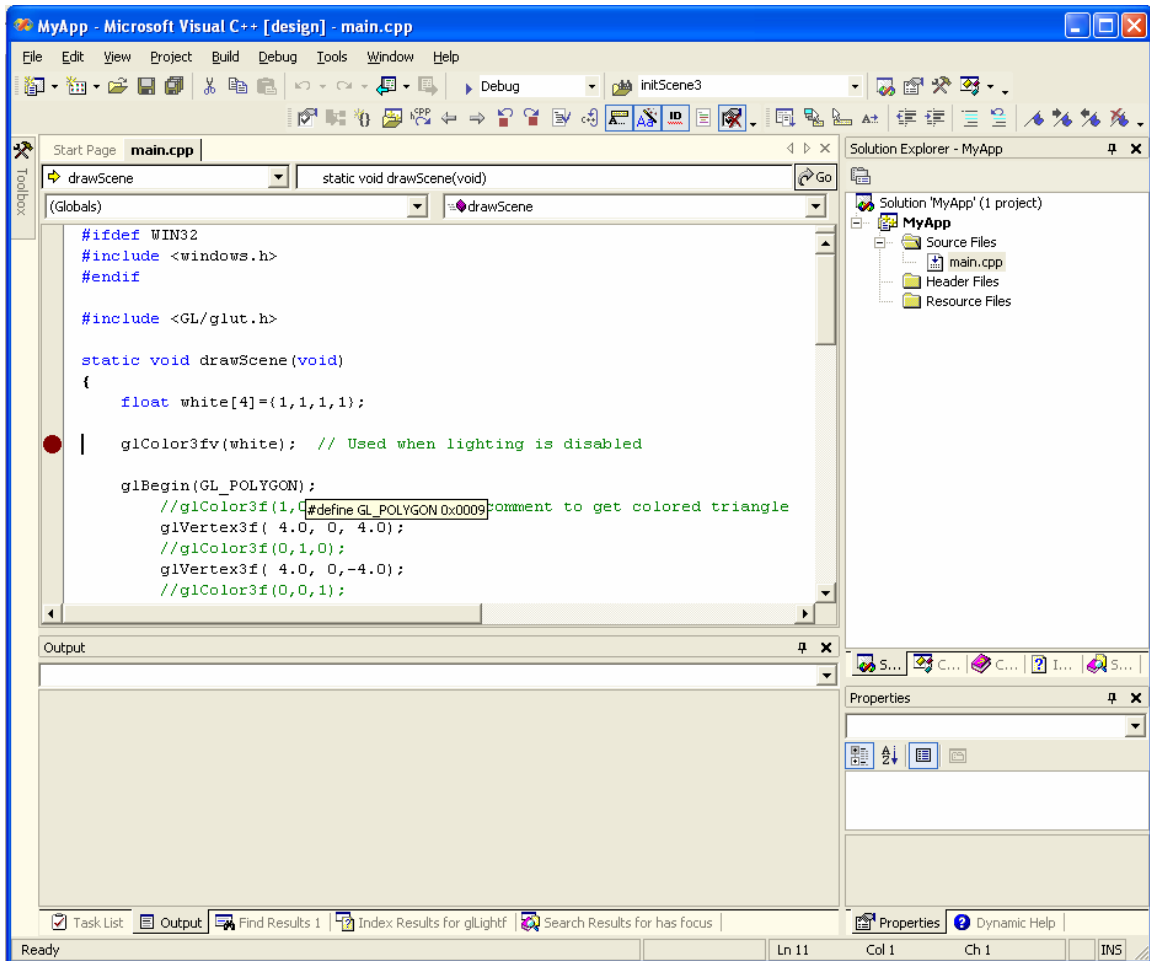


## Using the debugger

Using the debugger is very easy and is highly recommended.

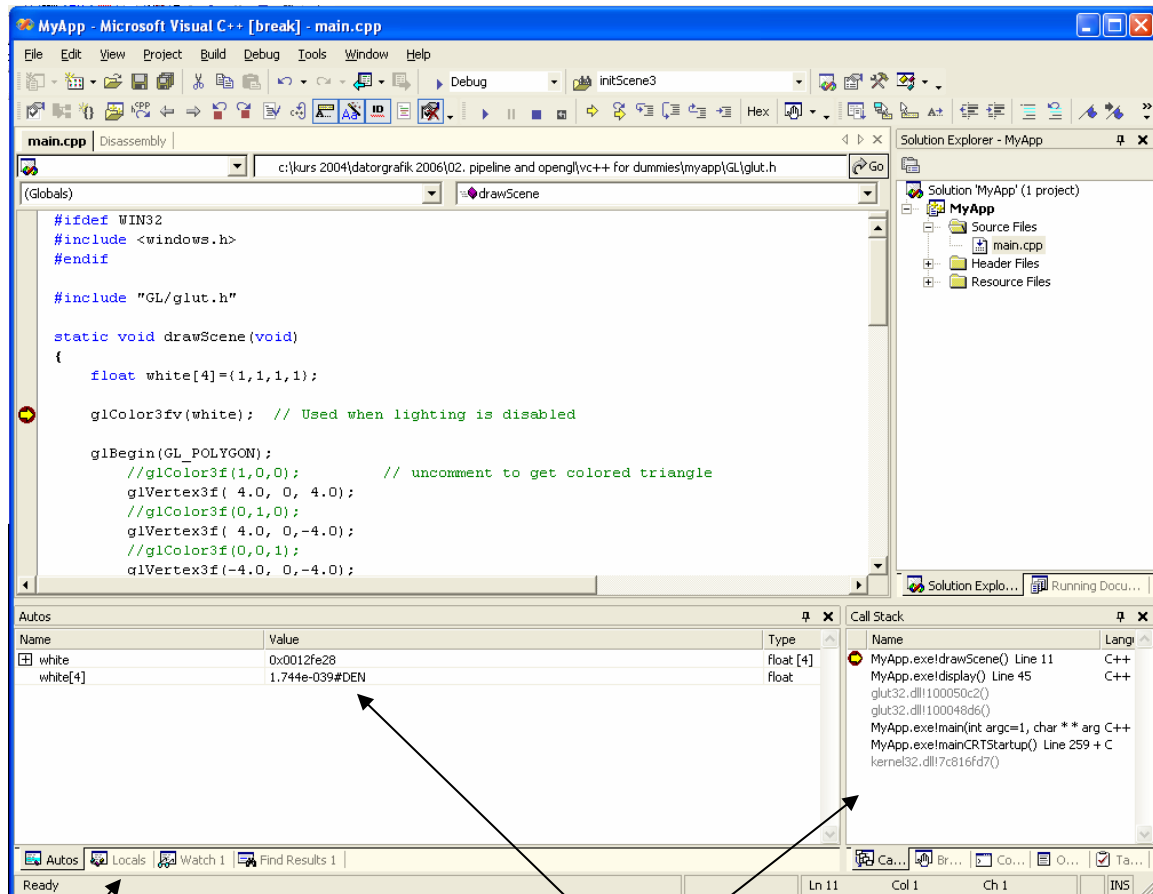
### **F9 sets/removes a break point.**

Set a break point by pressing F9 at the desired location in your source code. Remove the breakpoint by pressing F9 at the same location.





Press F5 to run the program and the debugger will stop at the location. Do this! It should now look like this:



The values of the most relevant variables are shown at tab “Autos” here. Note that the values can be modified directly by typing new values.

The values of all local variables are shown at tab “Locals” here:

The call stack is shown here:

### Summary of most common functions:

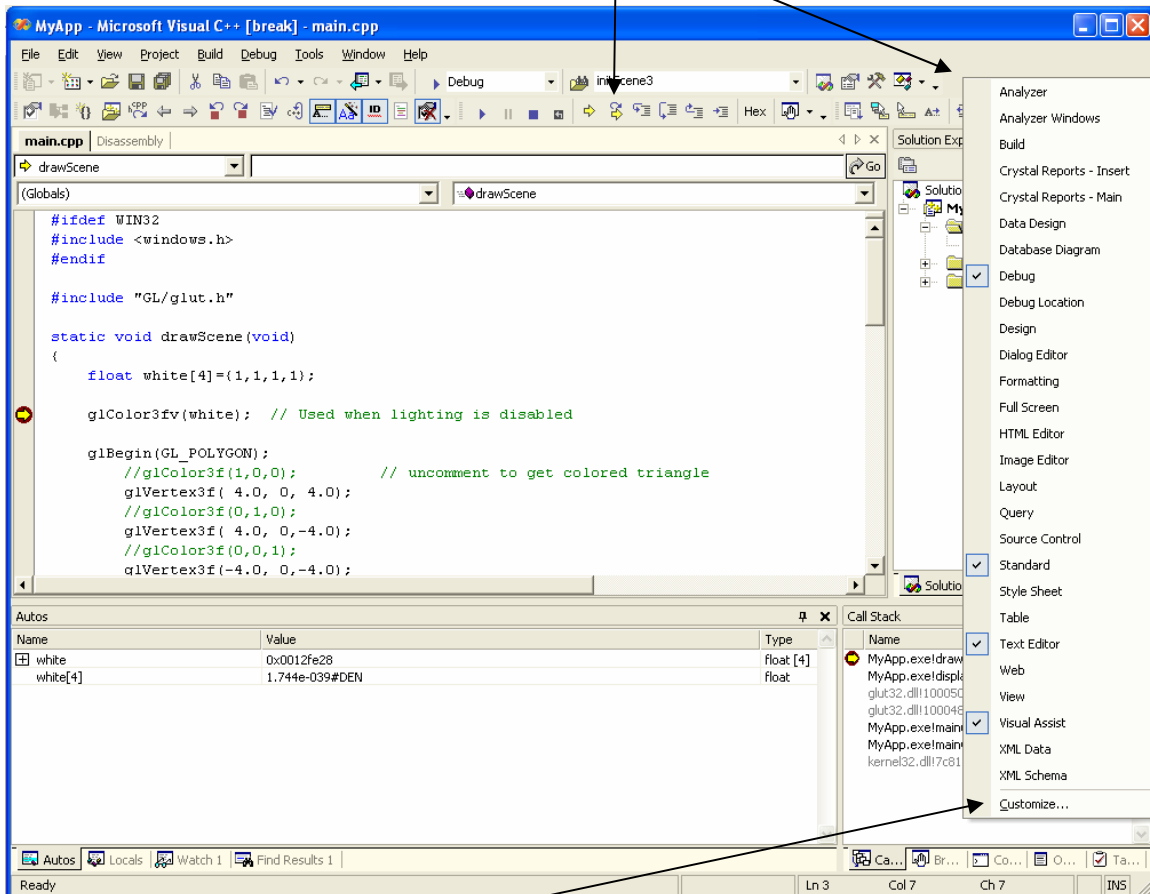
- F9 – sets/removes a break point at the location of the cursor
- F10 – steps one line in the source code
- F5 – continues the running of the program
- Shift + F5 – stops debugging
- F11 – steps into a function call if a function call is made at that line and source code for the function is available
- Shift + F11 – steps out from a function call

## Tricks

### Inserting the “Set Next Statement”-button.

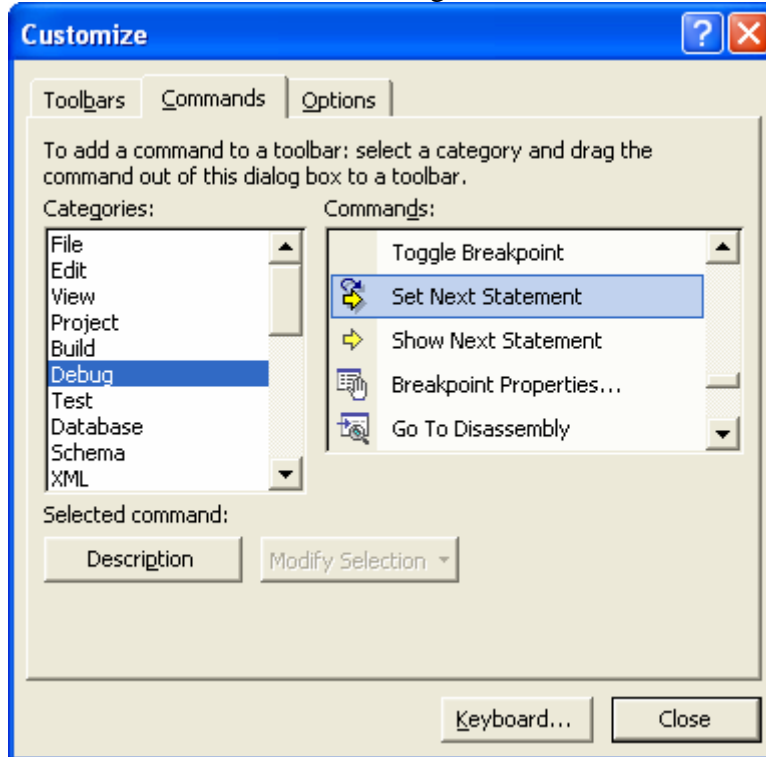
This is a button that when pressed, sets the current program position at the location of the cursor. This is unfortunately not available by default and it can be a very handy tool.

Right click at an empty position in the tool bar

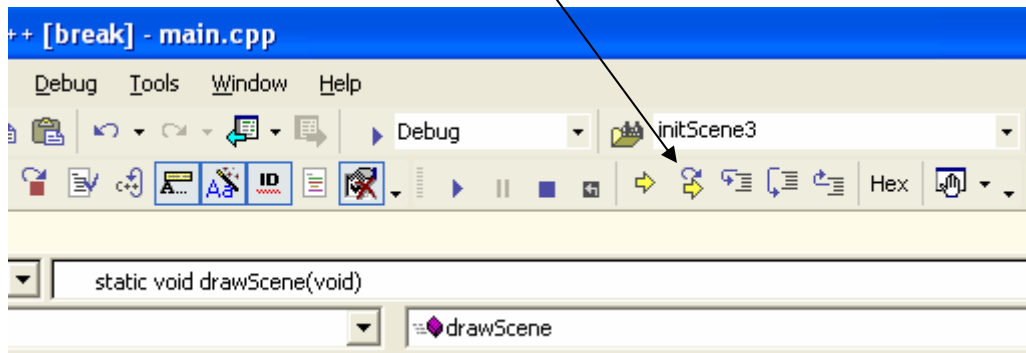


Select Customize...

Select tab “Commands” -> Debug -> Set Next Statement, as shown below.



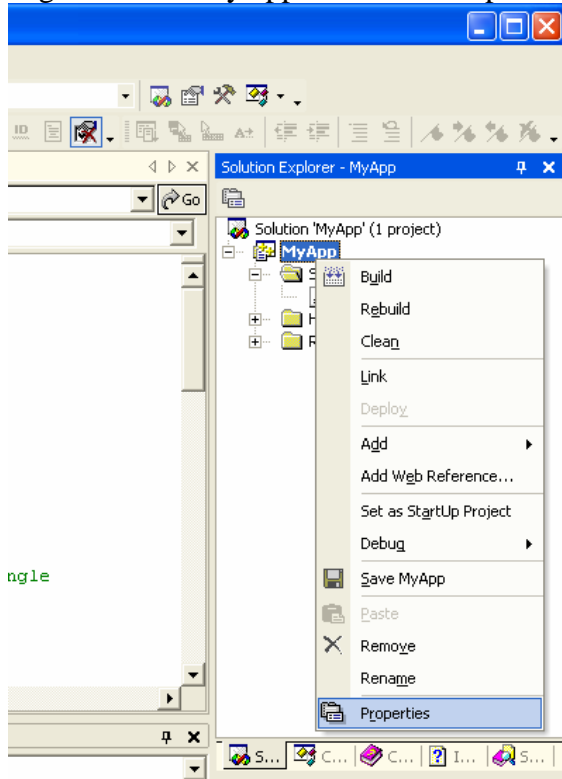
Now, use drag-n-drop by simply dragging the blue rectangle to the toolbar, where you want the button. My suggestion is here:



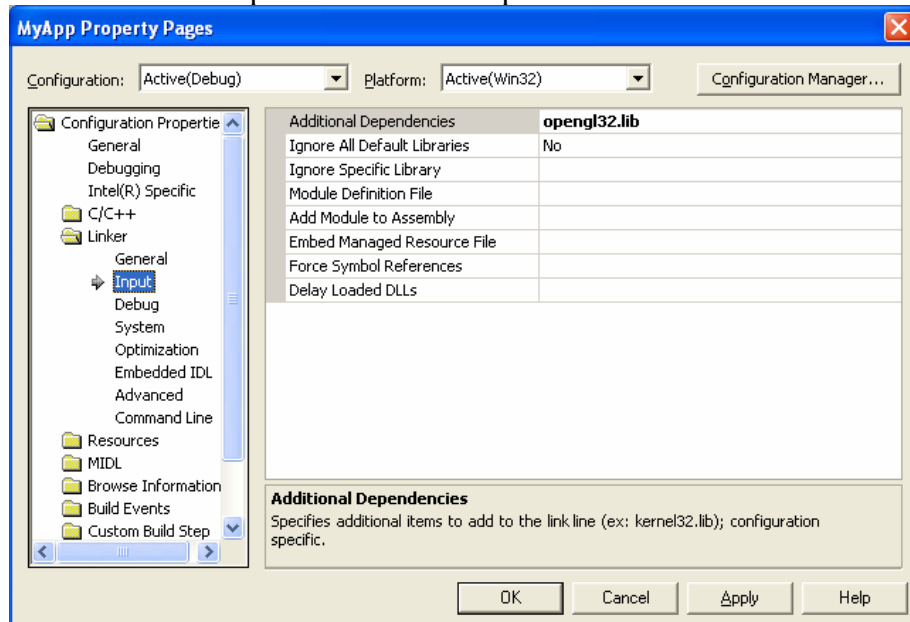
By pressing this button, the point of execution jumps to the position of the cursor ☺.

## Including lib-files

Right click on MyApp and select Properties:



Select “Linker->Input->Additional Dependencies”:

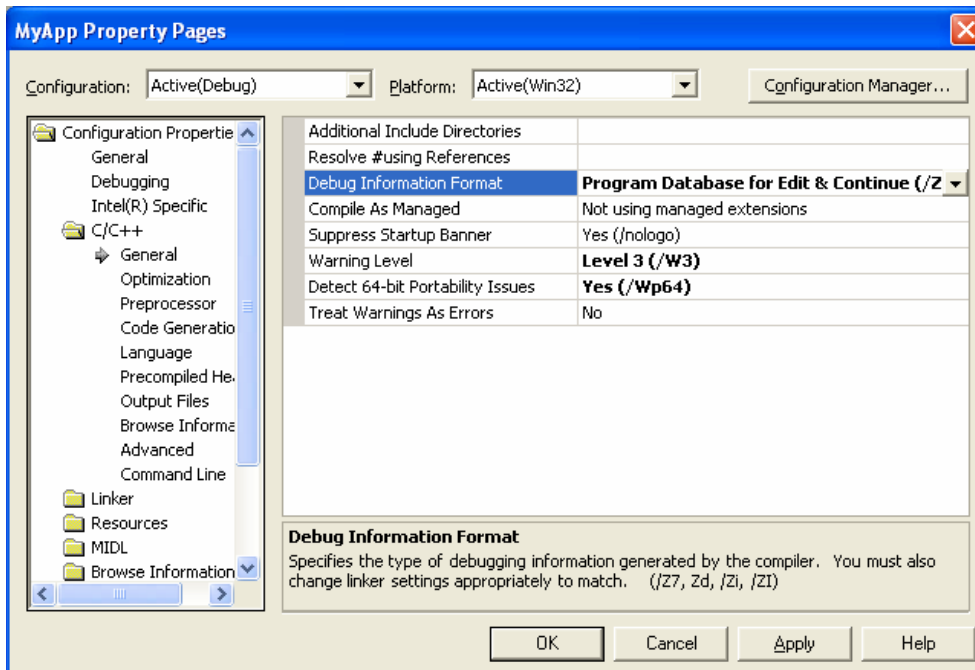


You can see that opengl32.lib is added. Add other libs to this field that you might want to use.

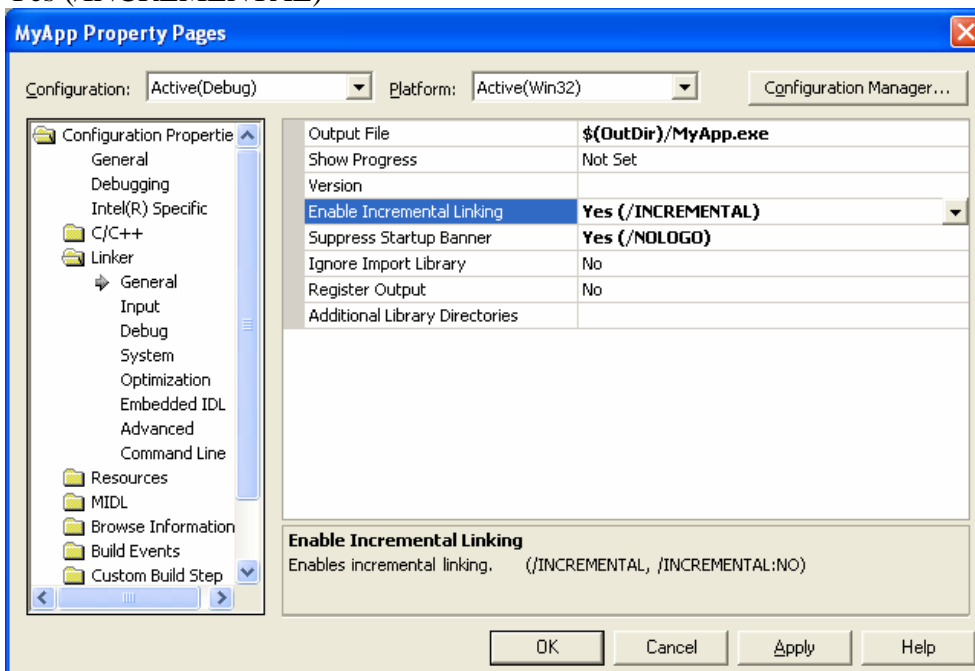
## Enabling Edit and continue

Edit and continue enables you to do small modifications to your code while you are debugging. This is enabled by default. In case it is disabled, enable it as follows:

In the same Property-dialog as above (i.e., right click on MyApp and select Properties), assert that C/C++ -> General -> Debug Information Format is set to **Program Database for Edit & Continue**

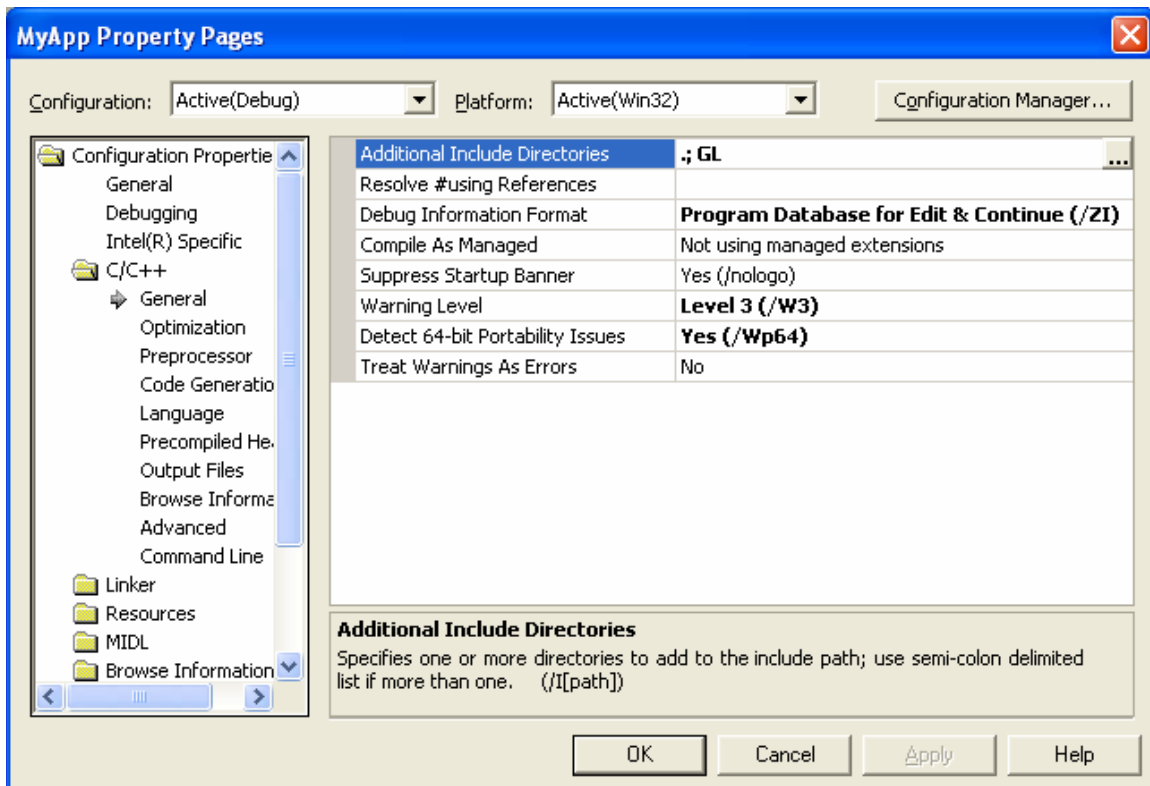


Furthermore, make sure that Linker -> General -> Enable Incremental Linking is set to Yes (/INCREMENTAL)



## Adding Include Paths

In the same Property dialog as above, select C/C++ -> General -> Additional Include Directories. I suggest that you add “.” Here I have also added the folder “GL” for purpose of illustration, as shown below. The “;” is used as a separator.



Comments:

“.” means that your project directory is added to the search paths for include files.

Note the difference between using "GL/glut.h" or <GL/glut.h> to include glut.h  
`#include "GL/glut.h"` searches for GL/glut.h in the project directory (H:\MyApp) and also in the Additional Include Directories and standard include directories, where stdlib.h, stdio.h, math.h etc are located.

However,

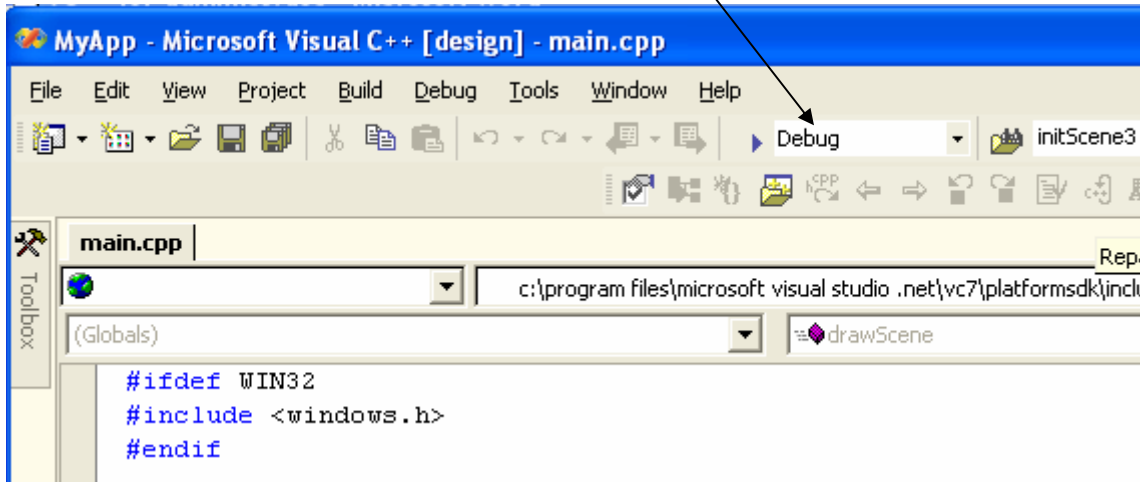
`#include <GL/glut.h>` skips the project directory and only searches for GL/glut.h in the Additional Include Directories and standard include directories.

Using " " is thus more general. By adding “.” to the Additional Include Directories you often avoid problems.

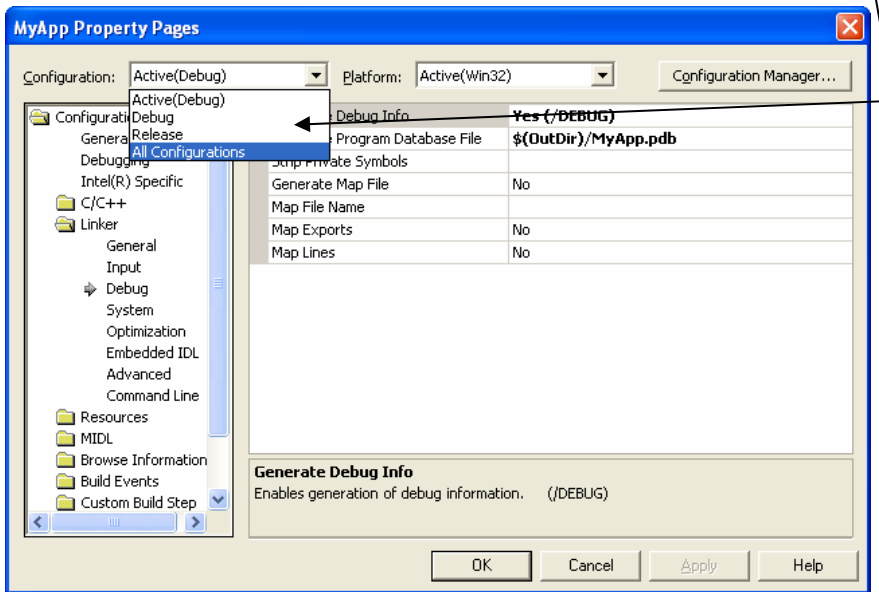
## Release or Debug Configuration

You can control whether a Debug-version or a Release-version of your code is compiled by toggling the active configuration here.

The release version is typically a lot faster, but by default lacks debugging capabilities such as break points and stepping. (However, in the Property dialog (see above), you can set “C/C++ -> General -> Debug Information Format” and “Linker -> Debug -> Generate Debug Info” to get debug information even in release mode.



Note that the settings in the Property dialog are individual per configuration (Release-mode or Debug-mode) including the settings for libraries, include paths etc. You can, however, modify the settings for Release and Debug simultaneously in the Property dialog, by specifying that the change applies to All Configurations.



## Miscellaneous about C++

Typically you store the class definition in a `class_name.h` file and the implementation of the methods (functions) in a `class_name.cpp` file.

The class definition in the `.h`-file looks like this:

```
// The #ifndef avoids the problems with multiple includes of the
// same file. Otherwise, the compiler may complain that the class
// is already defined.
#ifndef YOUR_CLASS_NAME_AND_SOMETHING_H
#define YOUR_CLASS_NAME_AND_SOMETHING_H

class Example
{
public:
    // Regarding speed it is good to inline the
    // constructors if they are short. Here they are empty.
    Example() {}

    // And the same applies for destructors
    ~Example() {}

    bool Method1();

private:
    int m_a;
};
#endif //YOUR_CLASS_NAME_AND_SOMETHING_H
```

The `class_name.cpp` file looks like this:

```
// windows.h must be the very first file to be included. Otherwise
// lots of compile errors may appear.
#include "windows.h"
#include "class_name.h"
#include <stdio.h>

bool Example::Method1()
{
    printf("Hi");
    return true;
}
```