C++ Programs and Visual Studio 2008 (Version 9)

Document Information

1. Current revision: 2009-12-28
2. Location of latest revision: http://icarus.cs.weber.edu/~dab/studio.pdf
3. Corrections & Suggestions: dbrinkerhoff@weber.edu
4. Contents

Contents

C++ Programs and Visual Studio 2008 (Version 9) .................................................. 1
  Document Information ................................................................. 1
  Contents ........................................................................ 1
  Visual Studio Architecture .......................................................... 2
  Quick Checklist .................................................................. 2
Visual Studio / Integrated Development Environment (IDE) Operation .................................... 3
  Starting Visual Studio ................................................................. 3
    Start the IDE .................................................................. 3
  Creating and Configuring a New Project and Solution ...................................................... 3
    Create A New Project ............................................................ 3
    Configure the new project as a Console Project ......................................................... 4
    Configure the Project as Empty ........................................................................ 5
    Add A New File to the Project .................................................................. 6
  Adding Existing Files To A Project .................................................................. 7
    Add an existing file to an existing project ......................................................... 7
  Compile and Run the Program ................................................................ 8
  Basic Compile-Time Debugging ................................................................ 9
    General Guidelines ................................................................. 9
    Syntax Errors .................................................................... 9
    Linker Errors .................................................................... 9
    Unexpected End of File Errors ................................................................ 9
  Debug Vs Release Mode (Managing the Executable Size) ............................................. 10
    Switch between Debug to Release Mode ....................................................... 10
  Locating “Lost” Files .................................................................. 11
    Locating Project Files ................................................................ 11
    Project Consistency: “I Modified My File But I Didn’t Make Any Difference” ........... 11
  Multi-Project Solutions .................................................................. 13
    The Current Working Directory – The Location of Data Files ......................... 14
  Adding Line Numbers in the Text Editor ................................................................ 14
Command Line Operation ........................................................................... 15
    Opening a Command Prompt Window .................................................... 15
    Compile a C++ Source Code File ..................................................... 18
    Compiling and Linking Multi-File Programs ............................................. 18
Makefiles and The make/nmake Command ......................................................... 19
    Makefile Format ................................................................. 19
    How Make Works .................................................................. 19
    General Rules ..................................................................... 20
    Other Tricks ...................................................................... 20
Visual Studio Architecture

Figure 1. Solutions contain one or more projects

Figure 2. Studio organizes solutions and projects as folders and sub-folders; each project represents one program

Quick Checklist

Create a new project/solution
- Win32 Console Application (p. 4)
- Project location (p. 4)
- Project name (p. 4)
- Solution name (p. 4)
- Make the project empty (p. 5)

Add a new file to the project
- Choose a .cpp or .h file as is appropriate (p. 6)
- Name the file (p. 6)

Add an existing file to the project (this does not change the physical location of the file)
- Add the file to the project (p. 7)
- Find and select the file by name (p. 7)
Visual Studio / Integrated Development Environment (IDE) Operation

Starting Visual Studio

Start the IDE: Start → All Programs → Microsoft Visual Studio 2008 → Microsoft Visual Studio 2008

Creating and Configuring a New Project and Solution

1. Create a New Project/Solution
   a. File → New → Project (illustration at left)
      - OR -
   b. Press the New Project Button (illustration at right)
2. Configure the new project as a Console Project
   a. Click the “+” adjacent to “Visual C++ Projects” in the Project types panel
   b. Select “Win32”
   c. Select “Win32 Console Project”
   d. Enter a project / solution name
      i. the solution name becomes the name of the folder containing the project and all files
      ii. the “Solution Name” is automatically entered when you enter the project “Name” and will be the same as the project name (you may change the solution name but it is not necessary at this time)
      iii. the executable program is named after the solution (i.e., the solution name with a .exe extension)
   e. Enter the location of the project (this may be on a flash or external hard drive)
      i. it is very important to remember where you create this folder
      ii. it is convenient to locate this folder near the root
      iii. it is convenient to make this path short, without any spaces in the name
   f. Press “OK”

Figure 7. Configuring a Console Project

Figure 8. Partial file structure of a project / solution

Figure 9. Project files seen the Solution Explorer
3. Configure the Project as Empty
   a. Select “Application Settings”
   b. Check “Empty Project” (note: this is critical!)
   c. Press “Finish”

4. If the Solution Explorer is not visible
   a. View ➤ Solution Explorer (illustration at right)
5. Add A New File to the Project
   a. Project ➔ Add New Item (illustration at left)
   b. Note that selecting File ➔ New ➔ File will allow you to create a new file but it is not placed in the project
      - OR -
   c. Press the Add New Item button, which is added to the toolbar once a project is opened (illustration at right)

   d. Choose type and name file
      i. Verify that Visual C++ is selected
      ii. Select “C++ File (.cpp)” or “Header File (.h)” as is appropriate
      iii. Name the file (note that the studio supplies the extension – .cpp or .h – as needed)
      iv. Press “Add”
      v. This opens an editor window into which you may enter source code

Figure 12. Adding a new item through the menu

Figure 13. Adding a new item with the New Item button

Figure 14. Selecting the file type and naming the file
Adding Existing Files To A Project

1. Add an existing file to an existing project (useful for moving files between home and school)
   a. Press the ▼ button attached to the New Item button (second button from the left)
   b. Select “Add Existing Item” (Figure 15)
   - OR -
   c. Project (menu) → Add Existing Item (Figure 16)
   d. Navigate to the location of the file (e.g., on a flash drive)
      either with the “My Computer” icon on the left or
      by pressing the ▼ button at
      the top of the window
   e. Select the file name in the window
   f. Press the “Add” button
   g. This does not copy the file into the project directory – it creates a link to the existing file but leaves the file
      in its current location

Figure 15. Adding a file with the New Item button

Figure 16. Adding a file through the Project menu

Figure 17. Windows XP

Figure 18. Windows 7
2. Compile and Run the Program
   a. Compile: Build ➔ Build Solution – Rebuild <project name> also works well
   b. Run: Debug ➔ **Start Without Debugging**
   c. If you select Start Debugging, the console window will close, preventing you from seeing any results

![Figure 19](image1)

**Figure 19.** Compile the program with F7 or **Build Solution** or with **Rebuild**...

![Figure 20](image2)

**Figure 20.** Run the program **Without Debugging**
Basic Compile-Time Debugging

**General Guidelines**
1. Begin with the first error in the file and work down
2. Locate the line by number on which the compiler reports the error
   a. Look for the error on this line
   b. If an error is not found on this line, work backwards toward the top of the file (the error will never be below this line)
3. If you do not find a bug in this file and if you have included your own (i.e., non-system) header files, examine the header files for bugs (often missing a semicolon after a class declaration)
4. Modern compilers perform an operation called “error recovery.” When a compiler detects an error, it may skip a large amount of code until it “figures out where it is” and resumes syntax checking. This often results:
   a. The compiler reports a few errors in your program
   b. You fix the first error and recompile your code
   c. The compiler now reports many more errors than before
   d. This means that the compiler is now checking the syntax of previously skipped code: do *not put the original bug back into your code*

**Syntax Errors**
1. Caused by missing, superfluous, or incorrect punctuation (semicolon, brace, operator, etc.)
2. Caused by incorrect identifier (name): not declared, misspelled, wrong capitalization, etc.
3. Scroll to the top of the error-list in the Output pane (typically at the bottom of the screen)
4. Double-click on the specific error message

---

**Output**

```
1>Compiling...
1>MyProgram.cpp
1>functions.cpp
1>cl\c:\\\c\MySolution\MyProject\functions.cpp(6) : error C2065: 'my_message' : undeclared identifier
1>Generating Code...
```

**Figure 21.** Syntax error in file “functions.cpp” at line 6

**Linker Errors**
1. Typically caused by misspelled function names
2. Locate the “external symbol” and make sure it is declared and/or spelled correctly

---

**Output**

```
1>Copyright (c) Microsoft Corporation.  All rights reserved.
1>Linking...
1>LIN : c:\\c\MySolution\MyProject.exe not found or not built by the last incremental link performing full link
1>functions.obj : error LNK2011: unresolved external symbol "char * my_message" (My_message@12BDA)
1>cl\c:\\c\MySolution\MyProject.exe : fatal error LNK1120: 1 unresolved externals
```

**Figure 22.** Linker error: this occurs during “Linking” and the error message contains the letters LNK; identifier “my_message” in not declared or is misspelled

**Unexpected End of File Errors**
1. Typically caused by a missing closing symbol: double or single quote opened but not closed, parenthesis, brace, or bracket opened but not closed
2. This may be difficult to find: start match all opening symbols for matching closing symbols
3. To prevent this problem
   a. Adopt a clean, well-structured programming style
   b. Watch color coding in the text editor
   c. Compile often
**Debug Vs Release Mode (Managing the Executable Size)**

When in debug mode, the generated executable code is “instrumented” with debugging code, which greatly increases its size. This is appropriate for program development but not for code that is commercially released.

1. **Switch between Debug to Release Mode**
   a. Press the ▼ button in “Solution Configurations”
   b. Choose “Debug” (default) or “Release” as desired
   - OR -
   c. Build ➔ Configuration Manager
   d. The default configuration is “Debug” (appropriate for program development but produces a very large executable)
   e. Press the ▼ button in “Active Solution Configuration”
   f. Select Release or Debug
   g. Press “Close”
   h. When compiled, the executable is located either in the “Debug” or the “Release” folder depending on the selected mode

---

**Figure 23.** Selecting Debug/Release mode from the main menu

**Figure 24.** Launching the configuration manager

**Figure 25.** Selecting the compile mode through the Configuration Manager

**Figure 26.** Executable location based on compile mode
Locating “Lost” Files

1. Locating Project Files
   a. There is a tab at the top of the editor for each open file
      i. Hover the mouse pointer over the tab to see the full path to the file
      ii. Right-click the tab and select “Open Containing Folder”

   ![Figure 27. Opening the folder containing a source-code file](image)

Project Consistency: “I Modified My File But I Didn’t Make Any Difference”

When you move code from one computer to another (e.g., moving code from the lab to home and back), it is possible to create two files with the same name. You edit the file open in the text editor but compile the other file. The result is that (a) you get the same compile errors each time that you build the solution, or (b) changes that you make to your code do not appear when you run the newly built solution.

1. The first thing that you can do to prevent this problem is to understand the file structure that Studio creates and to remember where your source-code files are located
2. If you suspect that you are editing and compiling two different files
   a. Copy the contents of the currently open file to the clipboard
      i. Select the file in question – click the appropriate tab at the top of the editor
      ii. Select all of the code by pressing Control-A
      iii. Copy the selected code to the clipboard by pressing Control-C
   b. Close the current file
      i. Right-click the file’s tab
      ii. Select “Close” (see Figure 27)
   c. Open the file associated with the project
      i. In the Solution Explorer, right-click the name of the file and
      ii. Select “Open” from the popup menu
   d. If the code in this file is different than the code in the file just closed and if you want to discard the code in the open file
      i. Select all of the code by pressing Control-A
      ii. Delete the code by pressing DELETE or DEL
      iii. Copy the code from the clipboard into this file by pressing Control-V

   ![Figure 28. Opening a file associated with a project through the Solution Explorer](image)

Files and folders created by MS Visual Studio IDE

```
location
|ual
\---solution[1]
  | solution.ncb
  | solution.sln
  +---debug
  | project.exe[2]
  | project.ilk
  | project.pdb
  +---project
  | file.cpp[3]
  | project.vcproj
  | project.vcproj.computer.user.user
  |
  +---Debug
  | BuildLog.htm
  | file.obj
  | project.exe.embed.manifest
  | project.exe.embed.manifest.res
  | project.exe.intermediate.manifest
  | mt.dep
  | vc80.idb
  | vc80.pdb
|
\---Release[2]
  | BuildLog.htm
  | file.obj
  | project.exe.intermediate.manifest
  | mt.dep
  | vc80.idb
  | vc80.pdb
|
\---release[2]
  project.exe[3]
  project.pdb
```

Legend

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>the location (folder) of the project (p. 4)</td>
</tr>
<tr>
<td>project</td>
<td>the name of the project (p. 4)</td>
</tr>
<tr>
<td>solution</td>
<td>the name of the solution (p. 4)</td>
</tr>
<tr>
<td>file</td>
<td>the name of the source code file (p. 6)</td>
</tr>
<tr>
<td>computer</td>
<td>the name of the computer (not significant)</td>
</tr>
<tr>
<td>user</td>
<td>the login name of the person who created the project (not significant)</td>
</tr>
</tbody>
</table>

[1] The solution name can be different than the project name, but the IDE automatically names the solution the same as the project; you can change the solution name but there is no advantage to doing so unless you are creating a multi-project solution (pages 4 and 13)

[2] The release folders are only created if the project is built in release mode (page 10)

[3] The IDE creates many files for its own use; generally, you will only be interested in source code files and occasionally in the compiled or executable versions (note that more complex projects will contain more than one source code file– including .h files)
Multi-Project Solutions

Each project represents a complete program whose source code may span multiple files. This implies that a project must define exactly one main function. Each project is organized as a sub-folder under the solution folder.

![Figure 29. The Visual Studio file structure of a solution containing two projects](image)

Multi-project solutions may be used to organize related programs. For example, building a separate solution for each example program written in class is tedious; instead, build one solution (perhaps for a chapter) and build separate projects for each example.

1. Managing multi-project solutions:
   a. Create the first solution/project as described previously (step 1, page 3 through step 3, page 5) but give the solution and project distinct, appropriate names (Figure 30)
   b. To add additional projects to an existing solution: File → Add → New Project
   c. To add a new item to a specific project, highlight the project in the Solution Explorer then add the item
   d. The “Start Without Debugging” (Control-F5) menu item applies to the Startup Project
   e. To make a project the Startup Project, highlight the project in the Solution Explorer
      i. Project (menu) → Set as Startup Project
      ii. OR, right-click the project in the solution explorer → Set as Startup Project

![Figure 30. Naming the project and solution differently](image)

![Figure 31. Working with multiple projects within a solution: select the project to add new items or to set the startup project](image)
The Current Working Directory – The Location of Data Files

When a process/program runs, it has a “current working directory,” that is, it has a location in the file system. Whenever the program reads or writes a data file that was opened using a relative path name, the data file’s location is relative to the current working directory. The current working directory of a program executed through the IDE is the project folder.

![Figure 32](image.png) The relative location of data files when running a program from within the IDE

Adding Line Numbers in the Text Editor

To display line number in the text editor
1. Tools (menu) ➔ Options... (at the bottom of the popup menu)
2. Expand the “Text Editor” entry (click the symbol at the left of “Text Entry”
3. Select either “All Languages” or C/C++
4. Check the “Line numbers” checkbox
5. Press “OK”

![Figure 33](image.png) Displaying line numbers in the text editor
Command Line Operation

1. Opening a Command Prompt Window (configured to run the command line compiler)
      
      ![Figure 34. Windows XP](https://example.com/figure_34)

      b. This window has the command line version of the compiler in its PATH environment; it also sets INCLUDE and LIB environment variables to locate header files (the target of #include) and libraries respectively
      c. cd (change directory) to the folder/directory containing the source code and proceed

2. (Alternate to 1 above) Configure and use a Command Prompt window
   a. Open a “Command Prompt” window: Start ➔ All Programs ➔ Accessories ➔ Command Prompt
   b. Run the following command at the prompt (the quotation marks and percent symbols are required):
      
      "%VS90COMNTOOLS%vsvars32"
      
      c. This command configures only the current command prompt window: it sets PATH, INCLUDE, and LIB environment variables

      ![Figure 35. Windows 7](https://example.com/figure_35)
3. (Optional) Permanently set the PATH, INCLUDE, LIB, and (on Windows 7) LIBPATH environment variables to support the command line compiler. (NOTE: an error in the PATH environment variable can make it difficult or impossible to run installed programs. If you make a mistake, press the “Cancel” button.)
   a. Open the System Properties panel
      i. (Windows XP) Start ➔ My Computer ➔ View system information (at the left)
      ii. (Windows 7): Start ➔ Control Panel ➔ System and Security ➔ System ➔ Advance system settings
   b. Choose the Advanced tab and press the Environment Variable button (Figure 36)
   c. In Environment Variables pane, System variables sub-pane, select PATH and press Edit (Figure 37)
   d. An “Edit System Variable” pane is displayed (Figure 38) with the variable name and its current value
   e. Update the value of the PATH variable
      i. Place the mouse pointer in the “Variable value” and press the END keyboard button
      ii. type a single semicolon (;)
      iii. Enter the location of the command line compiler and dynamic libraries (all one line)
         Windows XP:
         C:\Program Files\Microsoft Visual Studio 9.0\Common7\IDE;C:\Program Files\Microsoft Visual Studio 9.0\VC\BIN;C:\Program Files\Microsoft Visual Studio 9.0\Common7\Tools;C:\WINDOWS\Microsoft.NET\Framework\v3.5;C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727;C:\Program Files\Microsoft Visual Studio 9.0\VC\VCPackages;C:\Program Files\Microsoft SDKs\Windows\v6.0A\bin
         Windows 7:
         C:\Program Files (x86)\Microsoft Visual Studio 9.0\Common7\IDE;C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\BIN;C:\Program Files (x86)\Microsoft Visual Studio 9.0\Common7\Tools;C:\Windows\Microsoft.NET\Framework\v3.5;C:\Windows\Microsoft.NET\Framework\v2.0.50727;C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\VCPackages
      iv. If there was an error, press Cancel, otherwise press OK
f. For each of the environment variables, INCLUDE, LIB, and LIBPATH, do the following
   i. In Environment Variables pane, System variables sub-pane press New (Figure 37)
   ii. A “New System Variable” pane is displayed (similar to Figure 38)
   iii. Enter the variable name (INCLUDE, LIB, or LIBPATH)
   iv. Enter one of the values below
   v. Press OK
   vi. After all value are entered, press OK on all open panes

Windows XP:

LIB
C:\Program Files\Microsoft Visual Studio 9.0\VC\ATLMFC\LIB;C:\Program Files\Microsoft Visual Studio 9.0\VC\LIB;C:\Program Files\Microsoft SDKs\Windows\v6.0A\lib

INCLUDE
C:\Program Files\Microsoft Visual Studio 9.0\VC\ATLMFC\INCLUDE;C:\Program Files\Microsoft Visual Studio 9.0\VC\INCLUDE;C:\Program Files\Microsoft SDKs\Windows\v6.0A\include

Windows 7:

LIB
C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\ATLMFC\LIB;C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\LIB;C:\Program Files\Microsoft SDKs\Windows\v6.0A\lib

INCLUDE
C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\ATLMFC\INCLUDE;C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\INCLUDE;C:\Program Files (x86)\Microsoft SDKs\Windows\v6.0A\include

LIBPATH
C:\Windows\Microsoft.NET\Framework\v3.5;C:\Windows\Microsoft.NET\Framework\v2.0.50727;C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\ATLMFC\LIB;C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\LIB
4. Compile a C++ Source Code File
   a. If using only old header files (example at left): cl hello.cpp
   b. If using the new header files (example at right): cl /EHsc hello.cpp
   c. Note that “EHsc” is case sensitive

```
// Old-style header files
#include <iostream.h>
int main()
{
    cout << "Hello World" << endl;
    return 0;
}
```

```
// New-style header files
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello World" << endl;
    return 0;
}
```

5. Compiling and Linking Multi-File Programs
   a. Compile each source code file separately (the /c option stops the compiler after the object file is generated – i.e., it prevents a premature link phase):
      cl /c /EHsc part1.cpp
      cl /c /EHsc part2.cpp
      cl /c /EHsc part3.cpp
      This produces three files: part1.obj, part2.obj, and part3.obj
   b. Link each object file to create the executable: cl part1.obj part2.obj part3.obj
   c. This creates an executable named part1.exe
Makefiles and The make/nmake Command

One of the advantages of working in an integrated development environment (described in the next section) is that the IDE manages the process of compiling and linking multi-file programs. This feature, however, grew out of an older utility named make, which provides the same functionality. The make utility (the Windows version is named nmake) reads a user-created text file that contains a description of how to build a program (i.e., a description of how a set of files should be compiled and linked to create a program) called a makefile. (The make utility and makefiles have more general uses but the discussion here is limited to building programs; furthermore, they may also be used with any language that has a command line version of the compiler, but only C++ programs are illustrated here.)

Makefile Format

Makefiles contain a series of rules. Each rule contains at least one line that contains a target and a dependancy. The target is the name of the file that the make utility will create; the dependancy-list is a set of files that must exist and be up to date before the target can be created. Rules also contain zero or more (but typically one) line that specifies the command used to build the target from the dependancy files. It is important to note that each command line must begin with a TAB character as illustrated below:

```
target : dependancy-list
  <tab>command-to-build-target-out-of-dependancy-list
```

When the make utility is executed without arguments, it searches the current directory for a makefile. On a Windows platform, the make utility searches for a file name makefile by default. On systems that distinguish between upper and lower case characters (e.g., Unix or Linux), it searches first for a file named Makefile and then for a file named then makefile. Makefiles may be given any legal system name but the name passed to the make utility with the -f or /f command line option. This permits having multiple makefiles in a single directory.

```
make -f project.2
or
nmake /f project.2
```

How Make Works

It is possible, and quite common, for a file name to appear as a dependency in one rule and as a target in a subsequent rule. In the following example, the files db.obj, comp.obj, io.obj, and tree.obj first appear in the dependency list of the first rule and then again as the target of one of the following rules.

```
db.exe : db.obj comp.obj io.obj tree.obj
  cl /o db.db.obj comp.obj io.obj tree.obj

db.obj : db.cpp db.h
  cl /c db.cpp

comp.obj : comp.cpp db.h
  cl /c comp.cpp

io.obj : io.cpp db.h tree.h
  cl /c io.cpp

tree.obj : tree.cpp db.h tree.h
  cl /c tree.cpp
```

When the make utility reads the above makefile, it builds the dependency tree illustrated below. The target of the first rule becomes the root of the tree; the other rules may appear in any order. The make utility works from the leaves toward the root of each sub-tree, invoking the command line version of the Microsoft compiler to create each target, which is the root of a sub-tree. A target or root is recompiled if its timestamp is older than the timestamp of any of its sub-trees.
Assume, for example, that all of the .cpp files are compiled to produce the illustrated object files and further that the object files linked to create the executable. If db.c is modified, the next time the make utility runs, it will find that db.obj is older than db.out, which causes it to recompile db.cpp. Following this, db.exe will older than db.obj, which causes it relink the object files together to recreate db.exe. If tree.h is modified, then io.cpp and tree.cpp are recompiled (both depend on this header file) and then all of the object files are relinked. The make utility only rebuilds files that become out of date when dependant files are modified.

**General Rules**

The makefile illustrated above may be simplified by using a general rule (in bold in the following example).

```makefile
.cpp.obj:
    cl /c $<
    db.exe : db.obj comp.obj io.obj tree.obj
    cl /o db db.obj comp.obj io.obj tree.obj
    db.obj : db.cpp db.h
    comp.obj : comp.cpp db.h
    io.obj : io.cpp db.h tree.h
    tree.obj : tree.cpp db.h tree.h
```

The general rule indicates that whenever an object file (.obj) is needed and a C++ source file (.cpp) is available, that the former can be created from the latter with the cl command. The string $< in the second line of the general rule is a macro or placeholder: it is replaced with the name of the C++ file when the compiler executes.

**Other Tricks**

It is sometimes convenient to have a rule whose target is not reference in the dependency list of the first rule. In this case, the target can be specified when the make utility is executed:

```makefile
.cpp.obj:
    cl /c $<
    db.exe : db.obj comp.obj io.obj tree.obj
    cl /o db db.obj comp.obj io.obj tree.obj
    db.obj : db.cpp db.h
    comp.obj : comp.cpp db.h
    io.obj : io.cpp db.h tree.h
    tree.obj : tree.cpp db.h tree.h

clean:
    del *.obj
backup:
    copy *.cpp f:\backup
```

The last two rules have no effect unless specifically invoked:

```
nmake clean
or
nmake backup
```
Finally, it is sometimes convenient to create a “dummy” rule:

.cpp.obj:
   cl /c $<

**all** : **db.exe** **prog.exe**

db.exe : db.obj comp.obj io.obj tree.obj
   cl /o db db.obj comp.obj io.obj tree.obj

db.obj : db.cpp db.h
comp.obj : comp.cpp db.h
io.obj : io.cpp db.h tree.h
tree.obj : tree.cpp db.h tree.h

prog.exe : prog.obj pqx.obj tree.obj
   cl /o prog prog.obj pqx.obj tree.obj

prog.obj : prog.cpp prog.h
   cl /c prog.cpp

pqx.obj : pqx.cpp
   cl /c pqx.cpp

The target **all** is never created but when the makefile is processed, the dependencies (**db.exe** and **prog.exe**) are always examined. If either of these files is out of date as described above, they are updated as needed. This makes it possible to build two or more executables from a single makefile.