Using Borland C++ Tutorial

(Last update Aug 21, 1998 by R. Tront)

This tutorial provides an introduction to using Borland C++. Borland C++ comes with several compilers/linkers/debuggers: some are for use from a (DOS) command line, and some are for use within a window-based Integrated Development Environment (IDE). The IDE provided is a single application that allows you to edit, project manage, compile, link, and debug your program files. This tutorial starts out with an explanation for first year computing students of command line environments and of writing programs targeted for running under a command line operating system like DOS. It then moves on to using the IDE to write command line programs that will run and can be debugged in a simple window, but it does not describe MS-Windows application programming. Along the way, it introduces proper programming of multi-module applications.

Introduction

The master operating system (OS) program of a computer is designed to accept operational commands of various types from users. In most operating systems, the user can interact with the OS via a simple command line interface. A command line interface uses simple text characters rather than bit-mapped graphics. The screen scrolls up as the user and computer write information to the bottom of the screen. The user is prompted on the left (often with just a “>”) to enter a command which (s)he must have memorized. The command often has parameters following it (usually not in brackets). A common command is to type the name of a program file, which the operating system takes as a request to start the application program contained in that program file.

Sometimes application programs can be command line based, yet use simple scrolling menus. In the case of an application, there is usually only a limited number of application operations a user might want at any point in the program, and they are scrolled out as a numbered list so the user doesn't have to memorize them. The user types the number of the choice (s)he wants. Most projects in Cmpt 275 use this technique as mouse-based applications are fairly complex to develop for lower division undergraduates.

A mouse-based application can have either of two interfaces. Screen-character-based applications allow the user to point the mouse at any character(s) on the screen. A good example is the old MS-DOS Edit application might have used. These are programmed by the developer to allow highlighting or writing of whole characters by users anywhere on the screen (not just at the bottom). They detect which characters(s) the mouse is pointing to when a mouse button is pressed/released. Unfortunately, the mouse cursor is a character-sized rectangle that moves in fairly coarse jumps.

Most modern operating systems now also have a Graphical User Interface (GUI) which allows the user to use a mouse to point with pixel (rather than character) resolution. Pixels are little screen dots much smaller than characters. This makes mouse cursor motion much smoother. Using pixel graphics also makes it possible to display icons and fancy character fonts. Microsoft Windows is a GUI interface to the underlying operating system. It is also a library of functions that an application program writer can call to give an application a GUI appearance. Writing applications to take advantage of a system's GUI features is quite difficult, but the resultant GUI application program is often beautiful to use. Unfortunately, modern software development tools such as Borland C++'s Integrated Development Environment (IDE) can be very sophisticated, with thousands of kinds of requests you might want to make of it. Even though the IDE is itself a GUI application, developing even simple programs with it requires significant mastery of this complex development tool.

Because of this, we instead first introduce you to developing simple C++ programs with the command line tools that come with Borland C++. Later in this document you will see how to use the GUI-based IDE to ease some of the tediousness of command line development, and to use the accompanying GUI run-time debugger. The latter allows you to single-step through your program as it executes, seeing what it does (possibly wrong) during each source code instruction. By the way, there is an old character-based debugger available too.

Using the Borland C++ Command Line Tools

Sign on to the ACS Assignment Lab PC using the instructions provided in other hand-outs from your professor. Also, follow those instructions to format a 3.5" HD floppy disk (so you will be able to store your work on this floppy between visits to the Assignment Lab). To get started with the command line Borland tools, you will need a command line interface to the OS. You could find and start a command shell provided by the operating system, but a shortcut to a Borland customized one is available. But, you have to know how to get to the directory containing the Borland programs and documentation.

1) The Borland programs and documentation are found in the m:\program directory. Start Windows Explorer by using the start menu path Start|Programs|Windows Explorer. Choose the disk drive called m:, then the subdirectory \program, then the program called b32tools. This starts a black Windows NT text console window. As a side note, files that your Instructor might leave you will be on a different drive, N:.

2) You should now have a black command line OS user session window. Change your session's 'current' drive to be your floppy in disk drive a: by typing:

```
a:
```

followed by the 'Enter' key (sometimes called the carriage return key). Now start the Dos command line Edit program by typing:

```
edit
```

If you have used Edit before, press the <Esc> key to get a blank screen on which you can enter your program. Alternatively, press any other key to see the "survival guide" (i.e. essential basics of using Edit).

3) Enter the following program. You don't have to enter the comments if you don't want, as this is just a little test program that I have commented for you. Use the arrow keys or mouse to move around and correct your program source.

```
//simple.cpp ----------------------------------------------------------------
#include <iostream.h>  /*Indicates file containing cin/cout syntax (needed
                         to check your use of cin/cout for correctness).*/
int main (void)        /*Defines the main program as receiving no
                         parameters and returning an completion status
                         integer to the OS upon program termination.*/
{
      int i;          //Defines variables.

      cout << "Enter a value for I, then press <Enter>: ";
      //Note: No end of line above, so user next enters data on the same line.
      cin >> i;         //Read the input stream (keyboard) for i.
      cout << "The value you typed in was: " << i << endl;
                        //Note appended end of line.
      cout << "This program will now end." << endl;
      return(0);        //Tell OS that this program ended ok.
}  //-------------------------------------------------------------------------
```

4) Save your program on the A: floppy disk using the editor's File|Save As menu command. Call it "simple.cpp". Don't forget to specify the drive by pre-pending an "a:" to the file name you are asked to supply, or by indicating A drive in the Dirs/Drives box before entering the name. Exit this editor by choosing the File|Exit command. Note: Experienced PC users might consider using the C:/temp directory instead of A:, but then they *must* remember to transfer the source *and project* files to floppy before leaving.

5) To compile your program, type the following:

```
bcc  -c  -v  simple.cpp
```

bcc is the 16 bit Borland C++ Compiler program. -c is a command parameter to tell the compiler to compile only. -v tells the compiler to put extra information into the resultant object file to help with later debugging. A file called simple.obj will be generated if there are no resultant compile errors. Any compile errors in your program will be listed by the compiler. If necessary, go back and run the Edit program to correct your program, File|Save it, and re-compile it.

6) The generated object file is put on in the 'current' directory/disk drive. To see the object file generated, use the directory listing command: dir a:

7) To generate an executable program, you must link your .obj file with other supporting program libraries like the input/output libraries which translate ASCII keyboard characters into binary integers for you. The command to do this is:

```
tlink  /v  c0s.obj  simple.obj, simple.exe,, cs.lib  bidss.lib
```

Note carefully where there are commas vs. where there are blanks in the command.

- The first parameter specifies putting debugging info into the resulting simple.exe for later debugging.
- c0s.obj is the C++ program initialization code needed to initialize variables, et cetera, in your program (note: the 0 is a zero). Unlike Pascal, C++ supports the concept of 'initialized' variables.
- simple.exe is the name you want for your resultant program (don't leave this out or your program will wrongly be named c0s.exe).

- The last two command parameters are run-time libraries which contain basic C++ features and the cin/cout object files your program needs to function. If you are using floating point variables, you may need to add two more libraries to the command (emu.lib and maths.lib).

8) To run your program, simply type: `a:simple.exe`
from the command line. You will then see the output from your program, and be able to enter any inputs your program requires.

9) To prove that your assignments work, you will have to hand in to your instructor an execution log script of a user's interaction with your program. In Windows 95 and NT, command line sessions have no menu bar. But they do have a system menu icon in the extreme top left corner that you can mouse click on. Before ending your program, left mouse click on the system menu icon, then on Edit, and then on Mark. This gives you a white rectangular character cursor which you can mouse drag to highlight a region of user interaction. Highlight the execution log of your program, and then choose SystemMenu|Edit|Copy. Paste this text into any word processor (e.g. Edit, Windows Notepad, Windows Wordpad), and print it out to hand in to your instructor as proof that your program works.

10) Make sure your floppy disk contains simple.cpp and simple.exe before you sign-off and leave the Assignment Lab. To check they are there, type: `dir a:`
To kill the black command line session window, type `exit`.

Using Command Line Tools for Multi-File Programs

You are now going to modify your simple.cpp program file.

1) Type: edit a:simple.cpp
and change it to look like the following program. The changes are to add the second "#include...", define more variables, and add the lines from the "//Now try..." down to "cout << "i_minus2...."".

```
//simple2.cpp ---------------------------------------------------------------
#include <iostream.h>  /*Indicates file containing cin/cout syntax needed
                         to check your use of cin/cout for correctness.*/
#include "module2.h"
int main (void)        /*Defines the main program as receiving no
                         parameters and returning an completion status
                         integer to the OS on program termination.*/
{
int i, result, i_plus1, i_plus2, i_minus1, i_minus2;    //Defines variables.

cout << "Enter a value for I, then press <Enter>: ";
/*Note: No end of line above, so user next enters data on the same line.*/
cin>>i;                  //Read the input stream (keyboard) for i.
cout << "The value you typed in was: " << i << endl;
                         //Note appended end of line.
//Now try the function calls.
incrFunction(i, i_plus1, i_plus2);
result = decrFunction(i, i_minus1, i_minus2);
//And finally print the results.
cout << "result = " << result << endl;
cout << "i_minus1 = " << i_minus1 << endl;
cout << "i_minus2 = " << i_minus2 << endl;
cout << "This program will now end." << endl;
return(0);               //Tell OS that this program ended ok.
}//-------------------------------------------------------------------------
```

Use the File|Save As operation to save the modified file in a:simple2.cpp. Using Save As allows you to not overwrite simple.cpp with the changes, and thus keep the old version too. Note the newly modified main program calls two functions implemented in a second file called module2.cpp.

2) Most modern languages check that the number and type of parameters in function calls (even calls to functions in separate source files) are correct. This is done by having the calling program check that each call complies with the function 'prototype' (or 'function signature') of the called function. In Turbo Pascal, each 'unit' has an 'interface' section containing the correct signatures. In other languages, the interface is usually in a separate file from the implementation of the external functions. This separate file in C++ is called a 'header' file and thus has a .h file name suffix. The writer of the external function implementation (i.e. library) often first writes the function prototypes and puts them in a .h file. (S)he documents the .h file prototypes with everything a calling programmer needs to know about what the

functions do and how to call them. This information is best put in the .h file, as often the source code of the implementation is not distributed to application programmers so the algorithms can be kept proprietary. Finally, the .h file is #included into both the implementation and calling modules. When compiling either, if the calls to the functions, or the implementation of the functions, do not comply with the signature of the function prototype, an error is generated which prompts the programmer to correct the inconsistency. Create the following interface and save it in a file named module2.h

```
//module2.h ------------------------------------------------------------
/*Header files (.h) usually contain the declaration of function prototypes.
This module contains two functions that are related as they both accept
an input parameter i, and compute output parameters that are
either 1 and 2 larger, or 1 or 2 smaller than the input parameter.
-------------------------------------------------------------------------*/
void incrFunction(int i, int & i_plus1, int & i_plus2);
/*This function has an integer input parameter i, and has two output
  parameters.  The "&" indicates parameters to be passed by
  reference, just like the keyword "VAR" is used in Pascal parameter lists.
  The first output parameter will be set by the function to
  1 greater than i, and the second to 2 greater than i.
  The function returns nothing (void) and is thus really a 'procedure'.
-------------------------------------------------------------------------*/
int decrFunction(int i, int & i_minus1, int & i_minus2);
/*This function takes an integer input parameter i, and has two output
  parameters.  The first output is 1 less than i, and the second 2 less than
  i.  The function returns a BOOLEAN indicating whether i was less than 100.
-------------------------------------------------------------------------*/
```

Save these two prototype declarations in a single header file named a:module2.h.

3) Now we must define the source code for the implementation of these functions. What these functions do need not be commented in this code as it is already documented in the interface file (module2.h). But comments put in an implementation file describe *how* the module/function algorithm(s) work, not what it does. For instance, the goal of each loop should be mentioned to assist readers of the implementation. (e.g. //search the array starting from the rear for the desired value"). Run Edit and enter the following function definition (i.e. implementation) code:

```
//module2.cpp-----------------------------------------------------------
/*This module implements the two functions described in module2.h
*/
#include "module2.h"
//-------------------------------------------------------------------------
void incrFunction(int i, int & i_plus1, int & i_plus2)
{
//Any local variables needed would be declared right here.
i_plus1 = i+1;   //Here is the function's algorithmic body.
i_plus2 = i+2;
}
//-------------------------------------------------------------------------
int decrFunction(int i, int & i_minus1, int & i_minus2)
{
//Any local variables needed would be declared right here.
i_minus1 = i-1;  //Here is the function's algorithmic body.
i_minus2 = i-2;
if ( i < 100 )   //If i is less than 100, return true.
{
      return(1); //C and C++ have no BOOLEAN type or constants.  Normally,
}                //  an integer=0 represents false, and otherwise means true.
else
{     return(0);}
}//--------------------------------------------------------------------------
```

Use File|Save As to save this code in a file named a:module2.cpp

4) Now, from the command line compile the two .cpp files using the following 3 commands:

```
a:
bcc -c -v  module2.cpp
bcc -c -v  simple2.cpp
```

If these operations succeed, you will have compiled your library module thus creating module2.obj on your floppy disk in drive a:. The compile of the new main program results in a:simple2.obj. Both of these compiles will check the module2.h file to confirm consistency with the interface. If either of the operations fail, you will have to edit the offending file(s) and correct your syntax error(s).

5) To link the modules with each other and with the start-up and run time library code, use the following link command:

```
tlink /v c0s.obj simple2.obj module2.obj, simple2.exe,, cs.lib bidss.lib
```

(Note: If you are using floating point variables, recall you may need to add `emu.lib maths.lib` to the end of the above command.)

This should create the executable program simple2.exe on your floppy in the a: disk drive.

6) Try running your new program by typing: `simple2` from the command line prompt.

Using the Borland IDE

From your experience in the previous part of this tutorial, you can see that command-line-based program development is inelegant. In addition to having to compile every source file individually, you have to specify every .obj file and library that needs to be linked, and do that every time you do a link! (And let's face it, a program usually must be built many times before you get all the bugs out of it). In a big program there might be 1000 different modules (this author used to work for a company that had 2500 module programs containing 1 million lines of C), and having to type these into the link command each time would be ridiculous.

Please realize that when you change one module (say adding a semi-colon you forgot), you do NOT have to re-compile every module in the program. In a big application with 1000 modules, this could take an hour. Really, you only need to compile the single changed source file, and then re-link the resulting .obj file with all the other still valid .obj files. (Actually, it is slightly more complicated than this: if the changed file is a .h file that is included in several .cpp files, you must re-compile all those files which 'depend' on the changed file). Using a GUI-based Integrated Development Environment allows you to specify the main and .cpp files used in the program **once** (visually), and set the build options and libraries used to generate the target executable program once. Then any time you need to build the program, you only have to select the build operation from a menu. Even all the .h file dependencies are figured out for you.

Let's get started:

1) To start the Borland GUI IDE, use Windows File Explorer to find and double click on m:\program\Borland C++).

2) Select from the IDE menu bar File|New|Project. Replace the default project path and name with "a:\simple2.ide" (without the quotes). Make sure the Target Type is Application, the Platform is Win32, and the Target Model is 'Console'. (If you are using Borland 4.52 at home, you might want to choose a target type of Easywin, a Platform of Windows 3.x, and a Target Model of Large). Now click on OK. (In the Assignment Lab, you will now get a message saying "Error: Cannot save project file: bcwdef.bcw". It is trying to save the existing default null project into the Borland install directory, which is a read-only directory which we do not want changed. This is not a problem so just click on OK. You might also have to do this for the file bcwdef.dsw.).

   Notice that a new project window with 4 nodes is now created and shows the main program's executable and main source file.

```
simple2 [.exe]
      simple2 [.cpp]
      simple2 [.def]
      simple2 [.rc]
```

   (Note: if you are opening an existing project with Project|Open, you may also have to select the View|Project command to see the project window).

   The last two nodes of the project tree are unnecessary for simple programs. RIGHT mouse click the .def node, and delete it. RIGHT click the .rc node, and delete it.

   The IDE will create a project 'window' exactly the same even if there had been no source and executables already on your floppy disk, as every project needs a main executable file and a main source file. It would not have actually created these files, but would have started this project window for you, which shows the beginning of a 'dependency tree'. A dependency tree illustrates which files are needed to build those hierarchically above it (i.e. those nearer to the root). The dependency tree is stored in a 'project' file with a .ide file type.

3) Since from the command line tutorial, you do have a main program file on the same disk and directory as the project, it does know about your main program. Double click on the text of the .cpp node in this tree to see the IDE open your main program in the IDE's editor. If you want, you can edit it further here and save it using this improved (GUI) editor.

4) Since C++ is not a module-oriented language like Pascal, Modula-2, or Ada, the main program does not know that it needs "module2.cpp". The #include "module2.h" statement within your main does not tell it this, only that it needs module2.h to check the syntax of the procedure calls in the main source file. So we have to tell the project that there are other source files (fortunately, we only have to do this once, no matter how many times we later have to link). With the mouse pointer on the text of the .exe node in the project window, click the RIGHT mouse button. Now left click on Add Node. On the a: drive, select your module2.cpp file, then click OK. This will create another node one level indented from the .exe file. If you don't like the fact that module2.cpp is above the simple2.cpp node, then with module 2.cpp selected (i.e. highlighted), press the <Alt> and down arrow keys simultaneously.

5) To compile and link the program and all its libraries, RIGHT click on the .exe at the root of your tree, and select Build Node. Wait for it to indicate the build status is successful, then clear the dialog window by clicking OK. You will probably see an additional message window showing a log of the build steps taken and any build errors. Also notice that building automatically adds info (e.g. code size) to some of the nodes in the project window. Now, any time you want to re-build your system after making changes, you could just do this (i.e. no typing of long command lines are needed each time).

6) The build also converts the .cpp nodes into parent nodes. To see the dependent nodes of the .cpp files, left click on the "+" icon to the left of each .cpp node. Notice that the Autodependency feature of the IDE found all the .h files that were directly or indirectly used by the parent. Notice that your main program has a lot of dependencies. Obviously it directly depends on module2.h and iostream.h, since your main 'included' them. But you can also see that iostream.h, in turn, 'includes' yet other .h files! (If you are at home and can't see these additional dependencies, then you have not followed the additional system administration suggestions at the end of this tutorial.)

7) You can run this program by selecting Debug|Run from the IDE menu bar, or by double left clicking on the .exe node in the project window, or by clicking the lightning bolt from the tool bar. The program will be run for you in a command line window.

8) If you are running a Win32 console program, unfortunately the program window will disappear as soon as the program ends, thus not allowing you to see the resulting program output. To avoid this, replace the last output statement in simple2.cpp with these two statements:

```
cout << "Press Enter to end this program:";
cin.ignore().ignore();
```

This causes the program to read past the first carriage return you typed earlier in the execution, and waits for you to type one final carriage return.

Before entering this last carriage return, you might want to use the console window's system menu to highlight and copy the execution session, which can be pasted into a word processor for handing in as proof your assignment works. (If you are using Borland 4.52 and an EasyWin command line session, you will find it more convenient to use the Print command on the Easywin system menu which will directly output an execution log to the printer).

Finally, if your program produces a long output, it might exceed the Win32 console window's size. If you are using Windows NT (not '95), you can expand the console window's buffer. As soon as the black console window shows up, click on its system menu, select Properties|Layout, then change the Screen Buffer Size Height to a large value like 500. This way, the last 500 lines of execution output will be kept available for you to mark/cut/paste and hand in to your instructor. On the other hand, if you are using Windows95, once you have debugged your program with the IDE debugger, you will have to retarget for EasyWin and use an EasyWin console.

9) One nice thing about the IDE is that it includes a 'make' feature. Make is a feature that will build, or *'partially'* build a system, without requiring a lot of typing each time you need to build. In fact, Make is smart enough to only re-compile the minimum subset of source files needed considering the changes you have made. For instance, if you edit module2.cpp and then 'make' (rather than build) the executable, it will not bother re-compiling your main program, as the old simple2.obj is adequate for the new link. This will reduce the time to regenerate the executable (especially when working on a slow floppy or with a huge number of program source files). On the other hand, if you edit module2.h, then 'make' the executable, it will re-compile both of your .cpp files because they both include (i.e. depend on)

module2.h, and its change may affect the resulting executable. Make accomplishes this magic by looking at the last modification date of each .h, .cpp, .obj, and .exe file in the dependency tree, and the dependency relationships between nodes in the tree. In fact, if you re-make a program that has just been built or made, make will do nothing as everything is up-to-date!

10) Try out the IDE a little more until you feel comfortable with it. Try creating another program by creating a project for it and writing some code. To end your first IDE development session, simply double click on the system menu in the top left corner of the IDE. In the Assignment Lab, you might now get a message saying "Error: Can't create O:\BC5\BIN\BCCONFIG.BCW". It is trying to create a file in the read-only Borland Install directory. This is not a problem so just click on OK.) Also, if you are using Windows 3.1 at home, you must kill any DOS sessions you started within Windows and the only way to do this is to type "exit" in the Dos session (double clicking on its "-" won't kill a windowed Dos session in Windows 3.1).

Using the Borland IDE Debugger

Note that the Borland 5.0 IDE debugger will only work for 32 bit Console or 32 bit GUI programs. It will not work for DOS programs or 16 bit Windows targets. On the other hand, if you have only Borland 4.52 at home, it's IDE debugger will only work on 16 bit Easywin or 16 bit Windows programs. (Easywin produces a 16 bit console window program, but as of 97/5/12 we have not got it running in the Assignment Lab yet under the new Windows NT 4.0 OS we have installed).

1) To use the IDE debugger, you have to decide whether you want to set some breakpoints in your program, then use the Debug|Run command to start the program at full speed. Or whether you want to execute the program one statement at a time. In either case, when the program is paused, you can ask the debugger to allow you to Debug|Evaluate the value of any variable. This is much better than adding output statements in your program. No one in industry adds output statements for simple debugging. All programmers take the 20 minutes it takes to learn about their debugger and use it instead. Also, adding output statements can make really sneaky bugs move, so using a debugger is much better.

2) You can even use Debug|Watch to see in a special window the values of variables as they change in your program. Watching is very important in C/C++, as pointer errors often cause variables to be modified by program mistakes, but the program doesn't fail or go crazy until it possibly much later uses that variable. i.e. the symptom appears in a much different location in your program than where the cause is located! If you know a variable or pointer has a wrong value in it, put a watch on it and run the program and see ***when*** the value changes to the wrong value. This change might not be happening during the execution of the section of your program that you thought, but in some earlier statement.

3) To use breakpoints, open an edit window by double clicking on the .cpp (or .h) file of your choice. Click on an executable statement, then select Debug|Add Breakpoint. You can even ask the debugger to only stop at the breakpoint on the 19th execution of that loop. You can select more than one statement as a break point so the execution will stop at either. Use Debug|Run to start the program which will then execute at full speed until it encounters one of the statements you marked as a breakpoint. Evaluate or watch the value of the variables that you think will help you debug your program. Use Debug|Run to get the program going again, or you can single step through the program one statement at a time as described below.

4) If instead of running at full speed, you can start or continue your program using the Trace Into command. Tracing is started by clicking on the toolbar icon that shows a dotted arrow hopping <u>into</u> a set of parentheses. The Trace Into feature allows you to execute your program one statement at a time, seeing what happens in the execution user window after each step. (Note: you may have to move or minimize some windows to see the user program's command line console or Easywin interaction session; it might be <u>*behind*</u> the Borland IDE!). Your current execution location in the program source is shown highlighted in blue by the debugger in a source code edit window. The highlighted statement has NOT been executed yet; it is the next one to be executed.

5) Tracing may trace into some library code in your program that you don't want to bother crawling through one statement at a time (e.g. Windows or C++ library code you didn't write and presume works correctly). In this case you can try Step Over which rapidly executes through whole called functions at a time (including your functions in Module2.cpp). Step Over is started by clicking on the toolbar button which has a dotted arrow hopping over/past a set of parentheses icon (which represent a function call). Finally, if you want to speed through the rest of the program to the end, remove the breakpoints and select Debug|Run.

6) To find out more about the IDE debugger, select Help|Contents|Debugger.

SFU and Home System Administration Hints - for staff, instructors, and home system administrators:

i) When installing Borland C++ in the assignment lab, the path to \bc5\bin must be added using a MAP command. When installing on an unnetworked PC, this path must be added to the autoexec.bat file's path.

ii) TLINK will not run out of memory if run from a regular Dos session. But if run from a Dos shell under Windows, it will run out of XMS memory. The dosprmpt.pif file must be edited using the Pif editor to change the XMS memory 'limit' to at least 4096 kB.

iii) The 3 "s" characters just prior to the periods in the tlink command customize the link to generate for the DOS "small" memory model. This must match the memory model used by the command line compiler (which defaults to small).

iv) Make the following changes to the IDE defaults so the IDE portion of the tutorial works as described. Set the IDE desktop to how you want it (e.g. no project or file open, full screen). Then make the following changes to the default settings:

Options|Project:

-Directories - change Source Directories include path to: o:\bc5\include; c:\bc5\include

and lib path to: o:\bc5\lib; c:\bc5\lib

-Compiler|Precompiled headers - check 'Use but Do Not Generate'.

-Make|Autodependencies - check Cache & Display

Options|Environment:

-Editor|File - uncheck Create Backup

-Preferences - uncheck autosave environment, desktop, project, and check autosave editor.

Then use the IDE Options|Save... command which saves these defaults to the files:
-bcconfig.bcd (environment defaults),
-bcwdef.bcw (project defaults),
-bcwdef.dsw (desktop defaults)
into the directory \bc5\bin (which must be writable). Copy these files to the assignment lab directory o:\bc5\bin\defaults.