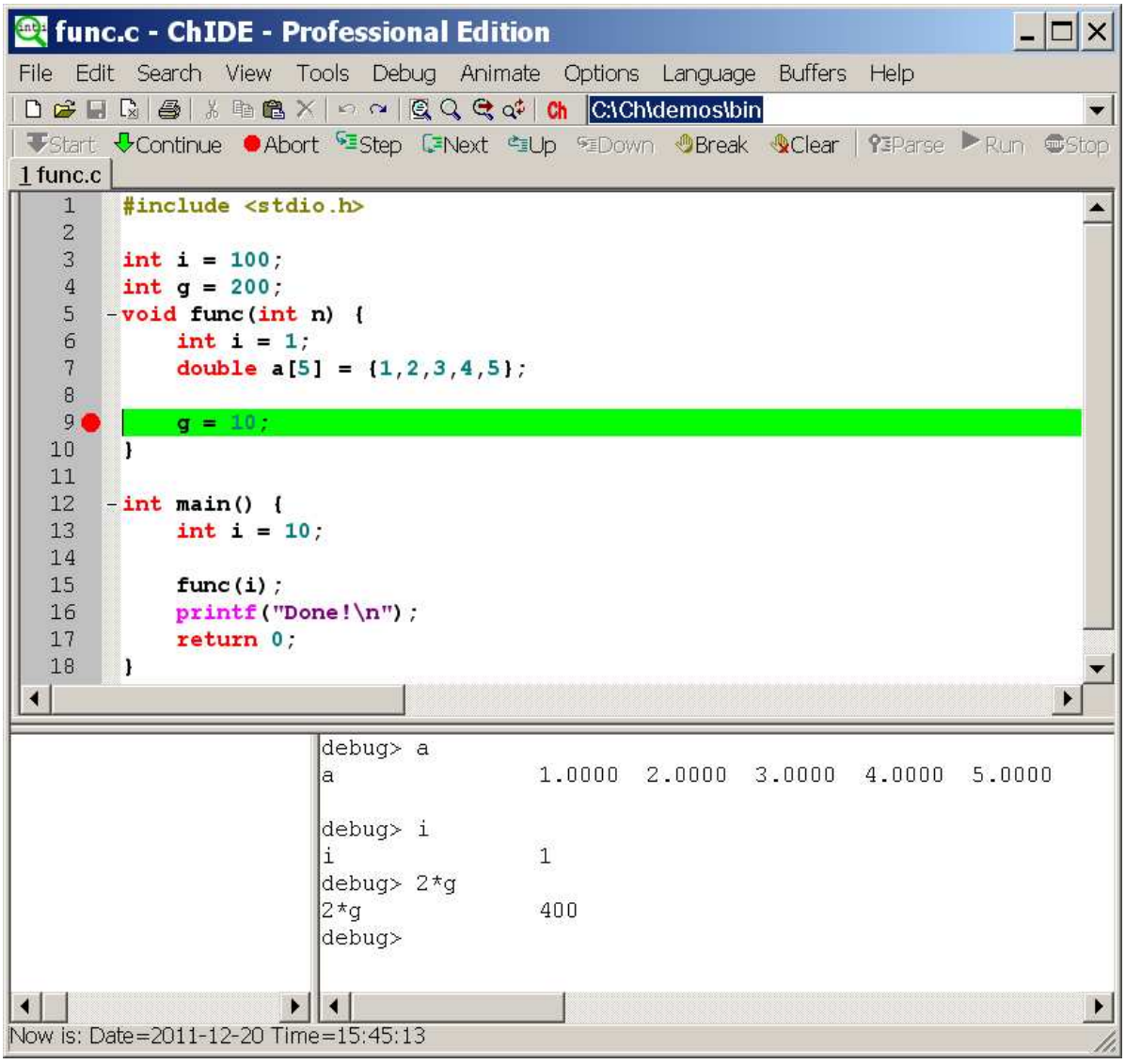
SoftIntegration®

Getting Started with ChIDE and Ch Command Shell

Ch Version 7.0



Copyright ©2012 by SoftIntegration, Inc., All rights reserved

How to Contact SoftIntegration

Mail	SoftIntegration, Inc. 216 F Street, #68 Davis, CA 95616
Phone	+ 1 530 297 7398
Fax	+ 1 530 297 7392
Web	http://www.softintegration.com
Email	info@softintegration.com

Copyright ©2012 by SoftIntegration, Inc. All rights reserved.
Revision 7.0.0, January 2012

Permission is granted for registered users to make one copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited.

SoftIntegration, Inc. is the holder of the copyright to the Ch language environment described in this document, including without limitation such aspects of the system as its code, structure, sequence, organization, programming language, header files, function and command files, object modules, static and dynamic loaded libraries of object modules, compilation of command and library names, interface with other languages and object modules of static and dynamic libraries. Use of the system unless pursuant to the terms of a license granted by SoftIntegration or as otherwise authorized by law is an infringement of the copyright.

SoftIntegration, Inc. makes no representations, expressed or implied, with respect to this documentation, or the software it describes, including without limitations, any implied warranty merchantability or fitness for a particular purpose, all of which are expressly disclaimed. Users should be aware that included in the terms and conditions under which SoftIntegration is willing to license the Ch language environment as a provision that SoftIntegration, and their distribution licensees, distributors and dealers shall in no event be liable for any indirect, incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of the Ch language environment, and that liability for direct damages shall be limited to the amount of purchase price paid for the Ch language environment.

In addition to the foregoing, users should recognize that all complex software systems and their documentation contain errors and omissions. SoftIntegration shall not be responsible under any circumstances for providing information on or corrections to errors and omissions discovered at any time in this documentation or the software it describes, even if SoftIntegration has been advised of the errors or omissions. The Ch language environment is not designed or licensed for use in the on-line control of aircraft, air traffic, or navigation or aircraft communications; or for use in the design, construction, operation or maintenance of any nuclear facility.

Ch, ChIDE, SoftIntegration, and One Language for All are either registered trademarks or trademarks of SoftIntegration, Inc. in the United States and/or other countries. Microsoft, MS-DOS, Windows, Windows 2000, Windows XP, Windows Vista, and Windows 7 are trademarks of Microsoft Corporation. Solaris and Sun are trademarks of Sun Microsystems, Inc. Unix is a trademark of the Open Group. HP-UX is either a registered trademark or a trademark of Hewlett-Packard Co. Linux is a trademark of Linus Torvalds. Mac OS X and Darwin are trademarks of Apple Computers, Inc. QNX is a trademark of QNX Software Systems. AIX is a trademark of IBM. All other trademarks belong to their respective holders.

Table of Contents

1 Introduction **1**

2 Executing C/Ch/C++ Programs in ChIDE **1**
2.1 Getting Started . . . 1
2.2 Editing and Executing C/Ch/C++ Programs . . . 2
2.2.1 Editing C/Ch/C++ Program Source Code . . . 2
2.2.2 Running C/Ch/C++ Programs and Stop Their Execution . . . 5
2.2.3 Output from Execution of Programs . . . 5
2.2.4 Detecting Program Syntax Errors . . . 6
2.3 Executing C/Ch/C++ Programs with the User Input . . . 7
2.4 Executing C/Ch/C++ Programs with Plotting . . . 7
2.5 Executing C/Ch/C++ Programs with Command Line Arguments . . . 9
2.6 Indenting C/Ch/C++ Programs . . . 14

3 Editing in ChIDE **14**
3.1 Browsing Files . . . 14
3.2 Edit . . . 14
3.3 Find and Replace . . . 15
3.4 Changing Font Size . . . 16
3.5 Folding . . . 16
3.6 Keyboard Commands . . . 16
3.7 Abbreviations . . . 17
3.8 Buffers . . . 21
3.9 Sessions . . . 21

4 Debugging C/Ch/C++ Programs in ChIDE **21**
4.1 Executing Programs in Debug Mode . . . 21
4.2 Setting and Clearing Breakpoints . . . 22
4.3 Monitoring Local Variables and Their Values in the Debug Pane . . . 22
4.4 Monitoring Variables in Different Stacks and Their Values in the Debug Pane . . . 22
4.5 Using Debug Commands in the Debug Command Pane . . . 24
4.6 Using the Debug Console Window for Input and Output . . . 29

5 Getting Started with Ch Command Shell **30**
5.1 Portable Commands for Handling Files . . . 31
5.2 Setup Search Paths for Commands, Header Files, and Function Files in Ch . . . 33
5.3 Interactive Execution of C/Ch/C++ Programs . . . 35
5.4 Interactive Execution of C/Ch/C++ Expressions and Statements . . . 35
5.5 Interactive Execution of C/Ch/C++ Functions . . . 38
5.6 Interactive Execution of C++ Features . . . 40

6 Interactive Execution of Commands in the Input/Output Pane **40**

7 Quick Animation **41**

8 Compiling and Linking C/C++ Programs in ChIDE **42**

9 Other Computer Languages Understood by ChIDE **45**

10 Local Languages Supported in ChIDE **45**

Index **46**

1 Introduction

Ch is an embeddable cross-platform C/C++ interpreter. It is a superset of C with classes in C++. It supports most new features in the latest C standard called C99 with other user friendly high-level extensions. Ch can be used for cross-platform scripting, shell programming, 2D/3D plotting, numerical computing, embedded scripting, and quick animation. With advanced numerical features, Ch can be conveniently used for various applications in engineering and science. However, Ch is especially suitable for interactive classroom presentations in teaching and for students learning C/C++.

An Integrated Development Environment (IDE) can be used to develop C and C++ programs. It can typically be used to edit programs with added features of automatic syntax highlighting and run the programs within the IDE. ChIDE is a cross-platform IDE to edit, debug, and run C/Ch/C++ programs in Ch interpretively without compilation. The user can set breakpoints, run a program step by step, watch and change values of variables during the program execution, etc. ChIDE is developed using Embedded Ch. It is the most user-friendly IDE for beginners to learn computer programming in C and C++. ChIDE can also be used to compile and link edited C/C++ programs using C/C++ compilers of your choice such as Microsoft Visual Studio .NET in Windows, GNU gcc/g++ in Linux and Mac OS X.

Because Ch is interpretive, C/C++ expressions, statements, functions, and programs can be readily executed in Ch without compilation. Therefore, Ch is an ideal solution for teaching and learning C/C++. An instructor can use Ch interactively in classroom presentations with a laptop to quickly illustrate programming features, especially when answering students' questions. Learners can also quickly try out different features of C/C++ without tedious compile/link/execute/debug cycles. To assist beginners in learning, Ch has been especially developed with many helpful warning and error messages when an error occurs, instead of cryptic and arcane messages like *segmentation fault* and *bus error* or crashing.

This brief document will get the user to quickly start using ChIDE and Ch command shell to learn computer programming and develop programs in C/Ch/C++.

2 Executing C/Ch/C++ Programs in ChIDE

2.1 Getting Started

ChIDE can be launched by running the same program **chide** across different platforms.

In Windows, ChIDE can also be conveniently launched by double clicking its icon shown in Figure 1 on the desktop.

In Mac OS X x86, ChIDE can also be launched by clicking the icon shown in Figure 1 on the dashboard or in the Applications folder.

In Linux, ChIDE can also be launched under the entry Programming Tools in the startup menu. The command

```
ch -d
```

will create an icon for Ch on the desktop. If Ch is installed with a ChIDE, an icon for ChIDE will also be created on the desktop.

The command line option for the command **chide** is as follows.

```
chide [-d directory] [filenames]
```

Figure 1: The ChIDE icon in Windows, Mac OS X, and Linux.

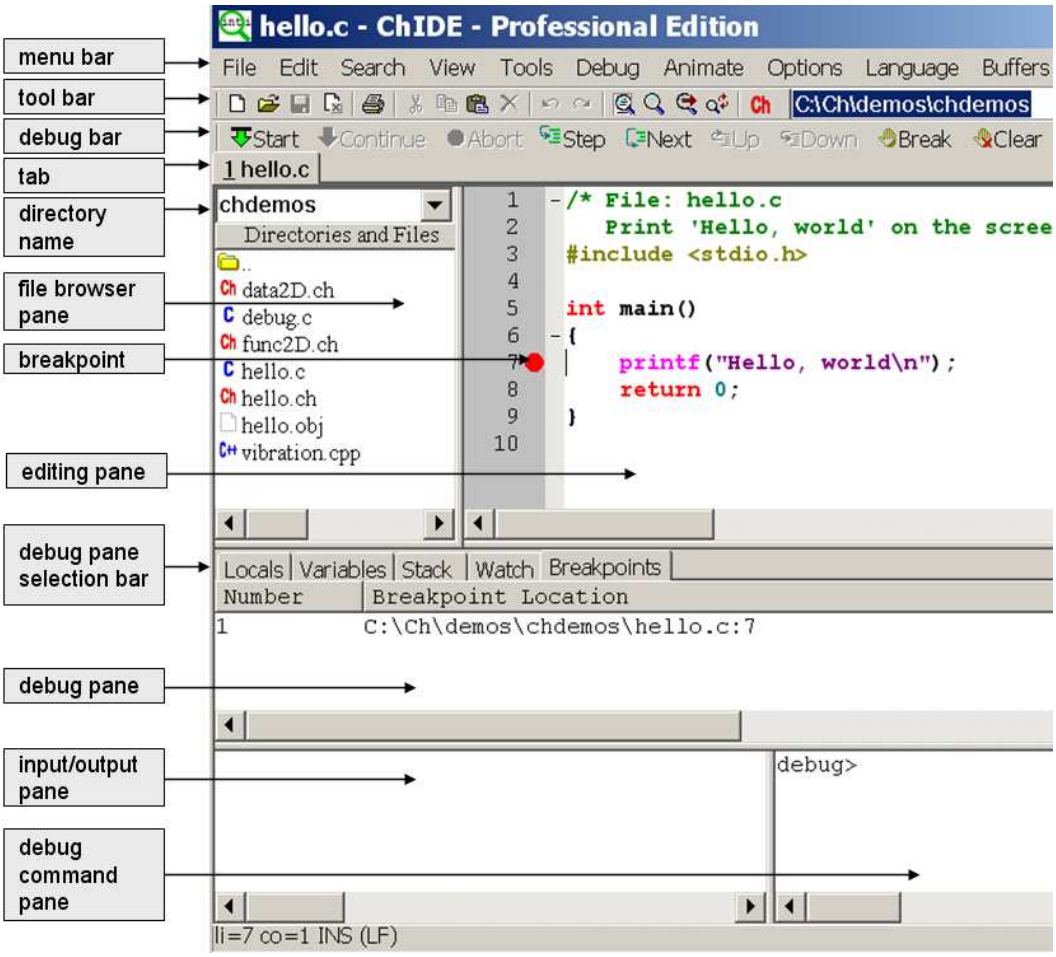


Figure 2: Terms related to the layout of ChIDE.

Without the command line option, the command **chide** will start ChIDE using the information previously saved when the ChIDE was exited. The option `filenames` will open one or multiple files. With the option `-d directory`, the file browser pane of the ChIDE will list files in the directory `directory`. For example, the command

```
chide -d /Users/home/harry file1.ch file2.ch
```

will open files `file1.ch` and `file2.ch` in the current working directory and the file browser pane will list files in the directory `/Users/home/harry`.

2.2 Editing and Executing C/Ch/C++ Programs

2.2.1 Editing C/Ch/C++ Program Source Code

Text editing in ChIDE works similarly to most Windows Mac text editors such as Word and Notepad with the additional feature of automatic syntax highlighting. ChIDE can hold multiple files in memory at one time but only one file will be visible. By default, ChIDE allows up to 20 files in memory at once as described in section 3.8. The general layout of ChIDE is displayed in Figure 2, which also shows various terms used to describe ChIDE in this document.

As an example, open a new document by clicking the command `File| New` or the first icon on the tool bar, and type

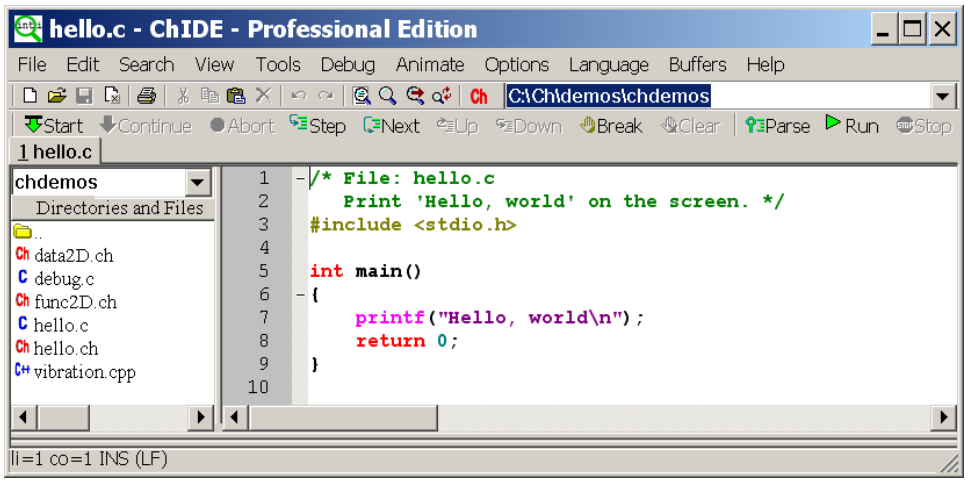


Figure 3: The program edited inside the editing pane in ChIDE.

```
/* File: hello.c
   Print 'Hello, world' on the screen */
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

in the text as shown in Figure 3 in the editing pane. The program appears colored due to syntax highlighting.

When ChIDE is started for the first time, programs in `CHHOME/demos/chdemos` will be copied to `HOME/chdemos`, where `CHHOME` is the home directory (folder) for Ch such as `C:/Ch` in Windows and `/usr/local/ch` in Mac, and `HOME` is the user's home directory such as `C:/Documents and Settings/Administrator`. The same program `hello.c` in `HOME/chdemos/hello.c`, such as `C:/Documents and Settings/Administrator/chdemos/hello.c` in Windows, can also be loaded using the `File->Open` command. By default, this program is loaded when the ChIDE is started. In Windows, a program listed under the Windows explorer can also be dragged and dropped on to the ChIDE, which will open the program in the editing pane.

The line numbers, margin, and fold margin on the left side of the editing pane can be suppressed as shown in Figure 4 by clicking the commands `View->Line Numbers, Margin, Fold Margin`, respectively. Clicking these commands again will bring these marks back. The fold point markers `'-'` and `'+'` on the fold margin can be clicked to contract and expand a fold for a block of code, respectively. In Figure 4, the file brower pane is closed by clicking the command `View->File Browser Pane`.

Save the document as a file named `hello.c` by the command `File->Save As`, as shown in Figure 5. You can also right click the file on the file name on the Tab bar, located below the debug bar, and then select the command `Save As` to save the program as shown in Figure 6.

There are five panes in ChIDE: the editing pane, file browser pane, debug pane, debug command pane, and input/output pane, as shown in Figure 2. The file browser pane is located on the left of the editing pane. The debug pane is located either to the below of the editing pane or on the right when CHIDE is displayed in vertical split to be described in the next paragraph. Initially it is of zero size, but it can be made larger by dragging the divider between it and the editing pane. The debug command pane is located either to the below of the debug pane or on the right. Details about the debug pane and debug command pane will be described in section 4. Similarly, the input/output pane is located either to the below of the debug pane or on the right. The input/output pane is on the left of the debug command pane. Initially the input/output pane

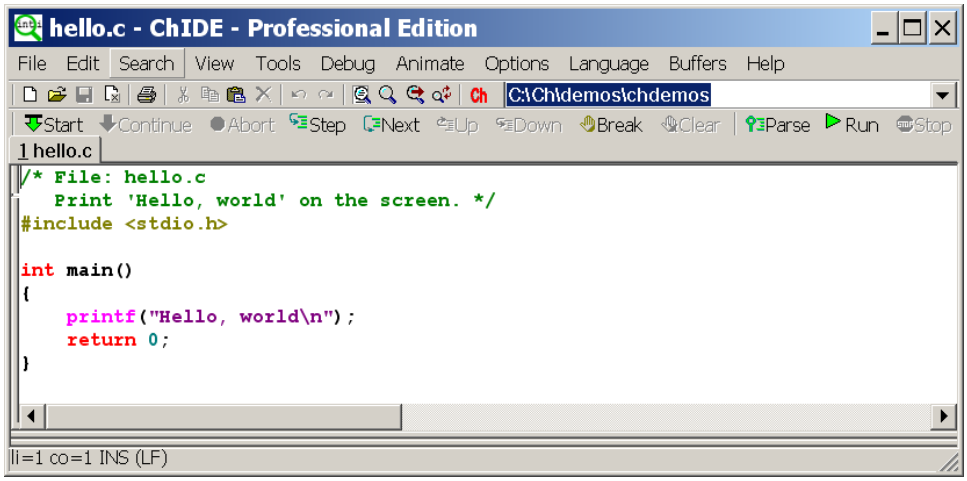


Figure 4: The program displayed without line numbers, margin, fold margin, and file browser pane in ChIDE.

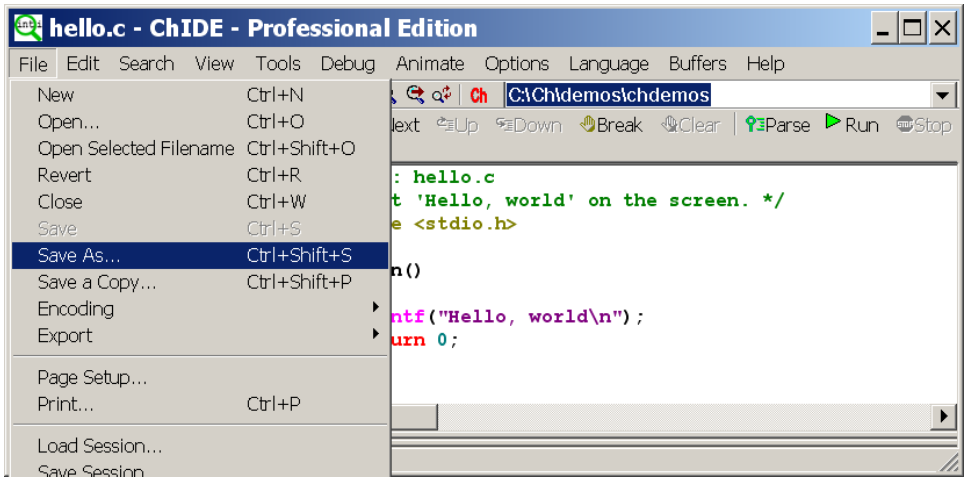


Figure 5: Saving the edited program using the command `File` | `Save As` in ChIDE.

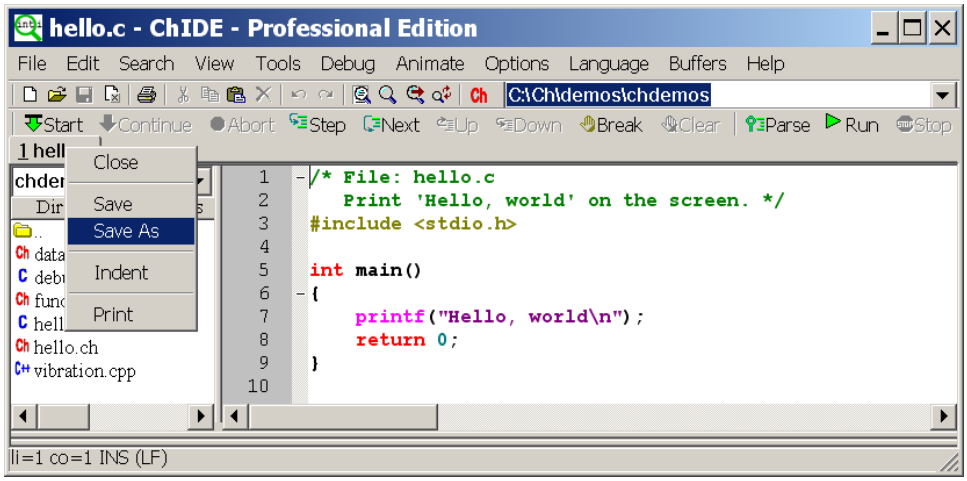


Figure 6: Saving the edited program in ChIDE by right clicking the file name.

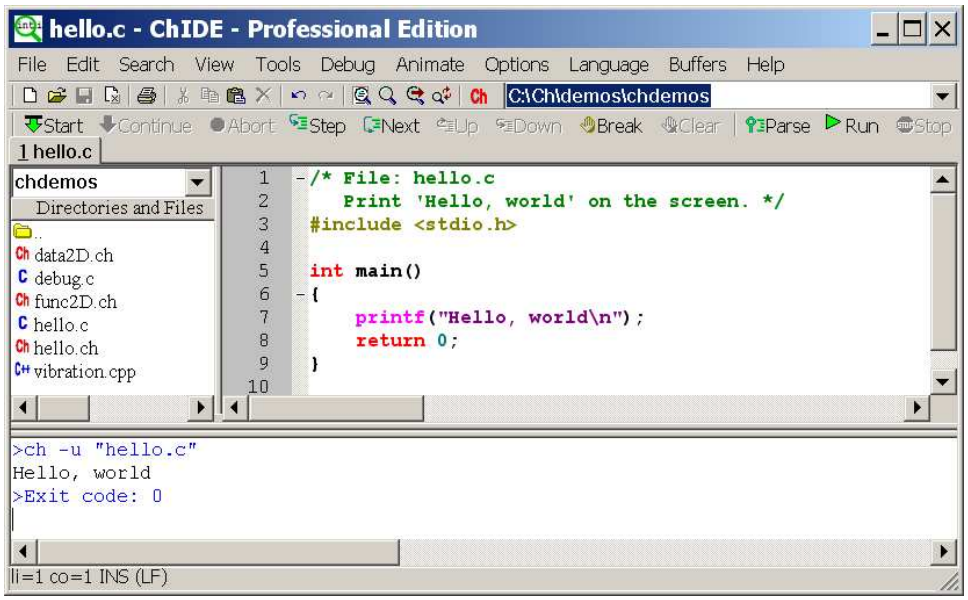


Figure 7: Executing the program using the command `Run` in ChIDE and its output.

is of zero size, but it can also be made larger by dragging the divider between it and the debug pane. By default, the output from the program is directed into the input/output pane.

The `View->Vertical Split` command can be used to change the layout of the ChIDE in vertical mode, in which the file browser pnae is on the left, followed by the editing pane, then the debug pane, and the input/output pane and debug command pane on the right. The location and size of the ChIDE, the sizes of file browser pane, editing pane, debug pane, and input/output pane in the current session are saved when ChIDE is closed. When ChIDE is started next time, these saved values in the previous session will be used for the new session. The command `View->Default Layout` will use the values in ChIDE global and user options files to reset ChIDE to use the default layout.

2.2.2 Running C/Ch/C++ Programs and Stop Their Execution

A C/Ch/C++ program with the file extension `.c`, `.ch`, `.cpp`, `.cc`, and `.cxx`, or without file extension can readily be executed in ChIDE. Perform the `Run` on the debug bar or `Tools->Run` command as shown in Figure 7 to execute the program `hello.c`. Instead of performing the `Run` or `Tools->Run` command, pressing function key `F2` will also execute the program. Details for keyboard commands are described in section 3.6.

You may use the command `Parse` on the debug bar or `Tools->Parse` to just check the syntax error of the program without executing it.

If the command execution has failed and is taking too long to complete, then the `Stop` on the debug bar or `Tools->Stop Executing` command can be used to stop the program.

2.2.3 Output from Execution of Programs

When the program `hello.c` is executed, the input/output pane will be made visible if it is not already visible and will display

```
>ch -u "hello.c"
Hello, world
>Exit code: 0
```

as shown in Figure 7. The first line in the blue color

```
>ch -u "hello.c"
```

from ChIDE shows that it uses the command **ch** to execute the program `hello.c`. The next line in the black color is the output from running the program `hello.c`. The last line in the blue color is from ChIDE showing that the program has finished. This line displays the exit code for the program. An exit code of 0 indicates that the program is terminated successfully by the statement

```
return 0;
```

or

```
exit (0);
```

in the program. If a failure had occurred during the execution of the program or the program is terminated with a non-zero value for a return or exit statement such as

```
return -1;
```

or

```
exit(-1);
```

the exit code would be -1.

2.2.4 Detecting Program Syntax Errors

ChIDE understands the error messages produced by Ch. To see this, add a mistake to the program by changing the line

```
printf("Hello, world\n");
```

to

```
printf("Hello, world\n";
```

Perform the `Run` or `Tools->Run` command for the modified program. The results should look similar to those below

```
ERROR: missing ')' before ';'
ERROR: syntax error before or at line 7 in file 'C:\ch\demos\bin\hello.c'
  ==>:    printf("Hello, world\n";
  BUG:    printf("Hello, world\n"; <== ???
ERROR: cannot execute command 'C:\ch\demos\bin\hello.c'
```

as shown in Figure 8. Because the program fails to execute, the exit code -1 is displayed at the end of the input/output pane as

```
>Exit code: -1
```

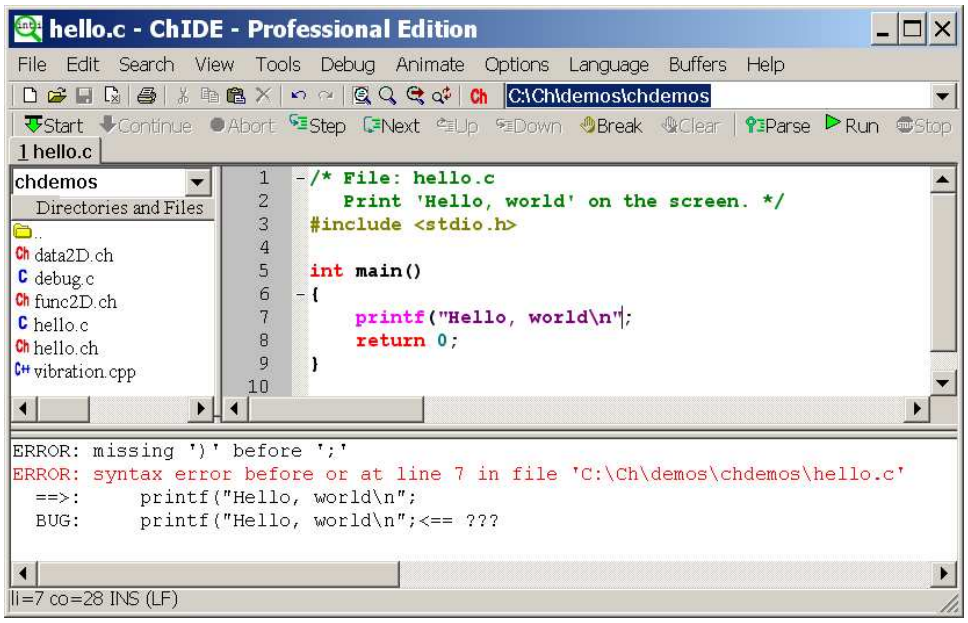


Figure 8: The error line in output from executing program `hello.c`.

If you double click the red colored error message in the input/output pane shown in Figure 8 with the left button of your mouse, the line with incorrect syntax and the error message in the input/output pane will be highlighted with yellow backgrounds as shown in Figure 9. The caret is moved to this line and the pane is automatically scrolled if needed to show the line. ChIDE understands both the file name and line number parts of error messages so it can open another file (such as a header file) if errors were caused by that file.

While it is easy to see where the problem is in this simple case, with a large file, the `Tools->Next Error Message` command, or the function key `F4`, can be used to view each of the reported errors. Upon performing `Tools->Next Error Message`, the first error message in the input/output pane and the appropriate line in the editing pane are highlighted with yellow backgrounds.

The command `Tools->Previous Error Message`, or the function key `Shift+F4`, can be used to view the previous error message.

The input/output pane can be opened and closed by the command `View->Input/Output Pane`. The contents of the input/output pane can be cleared by the command `View->Clear Input/Output Pane` or the function key F9 as shown in Figure 10.

2.3 Executing C/Ch/C++ Programs with the User Input

ChIDE can also execute programs that require the user's input through such C functions as **scanf()**. For example, load the program `C:/Ch/demos/bin/scanf.c` in Windows or `/usr/local/ch/demos/bin/scanf.c` in Linux or Mac OS X, as shown in Figure 11.

When the program is executed, the user will be prompted to input a number as shown in Figure 11. The user then must type in a number in the input/output pane. Both input number of 56 and output are shown in Figure 11.

2.4 Executing C/Ch/C++ Programs with Plotting

Running a C/Ch/C++ program with graphical plotting is the same as running other programs. This can be demonstrated by an example.

Type in the code as shown in Figure 12. The same program can also be loaded from `C:/Ch/demos/bin/fplotxy.cpp` When the program is executed, it creates a plot shown in Figure 13.

2.4 Executing C/Ch/C++ Programs with Plotting

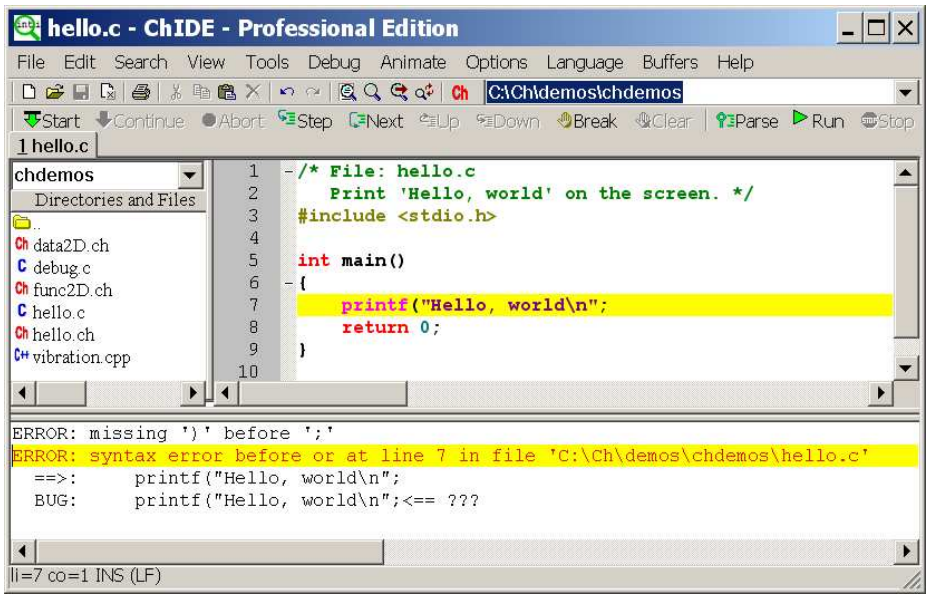


Figure 9: Finding the error line in output from executing program hello.c.

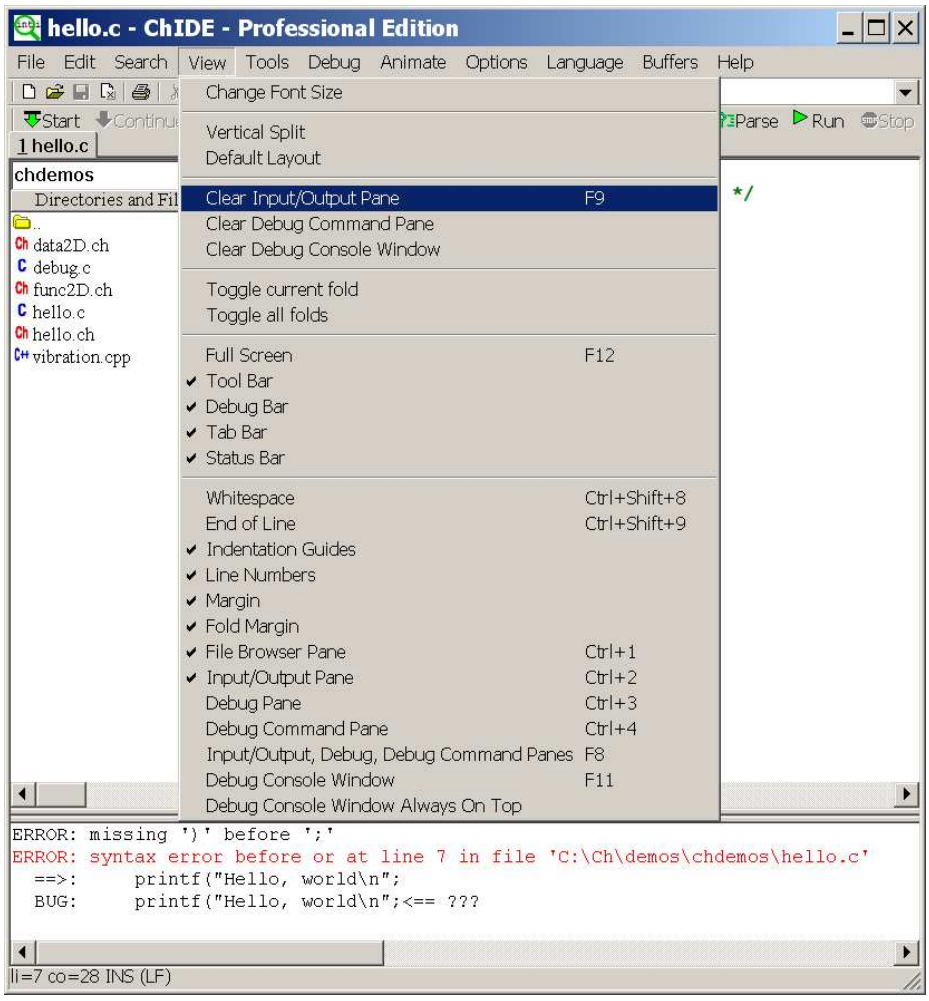


Figure 10: Clearing the contents in the input/output pane.

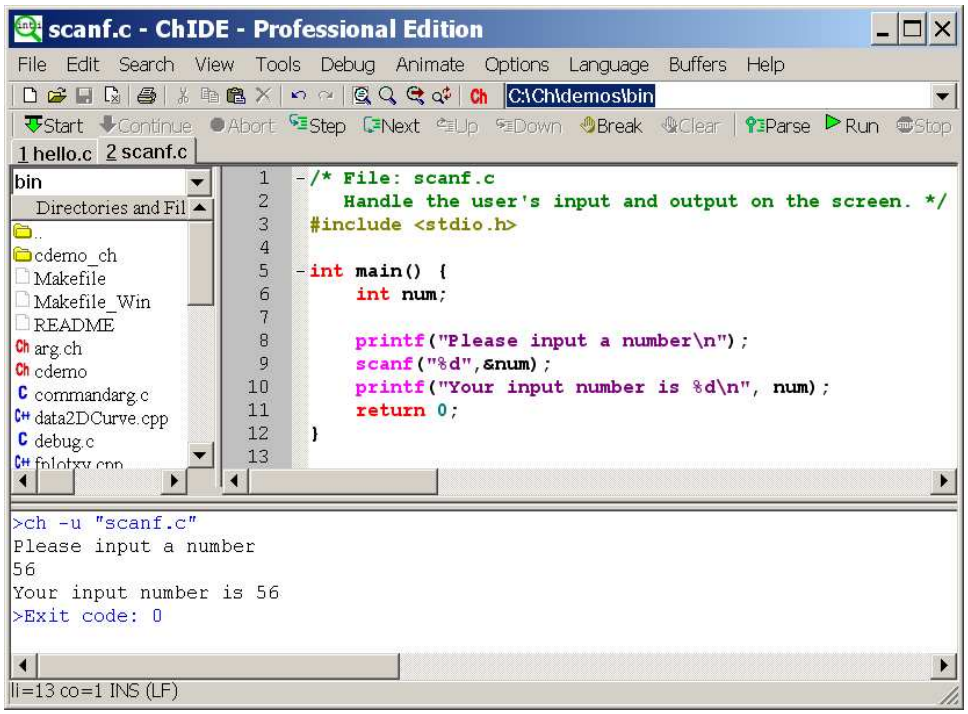


Figure 11: Executing the program with input and output.

The plotting function **fplotxy**() is available in Ch or SoftIntegration C++ Graphical Library (SIGL). The program uses the plotting function **fplotxy**() to plot function func() with 37 points and with the x value in the range from 0 to 360.

To compile a program using plotting features with header file **chplot.h**, the program has to be treated as a C++ program with file extension .cpp to link with a SIGL C++ plotting library. How to compile a C++ program using a C++ compiler will be described in section8.

Many sample programs are available in CHHOME/demos/bin and CHHOME/demos/lib/libch/plot directories to demonstrate capabilities and usages of the plotting features in Ch. For example, the program `C:/Ch/demos/bin/plotxy.cpp` uses the plotting function **plotxy**() plot data stored in arrays. When it is executed, it creates the same plot shown in Figure 13. The program `C:/Ch/demos/bin/fplotxyz.cpp` uses the plotting function fplotxyz() to plot the function $\cos(x)\sin(y)$ with two independent variables x and y for the x value in the range from -3 to 3 and y in the range of -4 to 4. The plot uses 80 points for both x and y coordinates. The program `C:/Ch/demos/bin/legend.cpp` shows how to add legends for multiple curves to a plot.

2.5 Executing C/Ch/C++ Programs with Command Line Arguments

ChIDE can run programs with changeable command line arguments. To set the command line arguments, use the `Tools->Command Line Arguments` command to view the modeless Command Line Argument dialog which shows the current command line arguments and allows setting new values. The accelerator keys for the main window remain active while this dialog is displayed, so it can be used to rapidly run a command several times with different arguments. Alternatively, a command executed in the input/output pane as described in section 6 can be made to display the modal Command Line Arguments dialog when executed by starting the command with a `'*'` which is otherwise ignored as shown below.

```
* C:/Ch/demos/bin/commandarg.c
* "C:/Ch/demos/bin/commandarg.c"
```

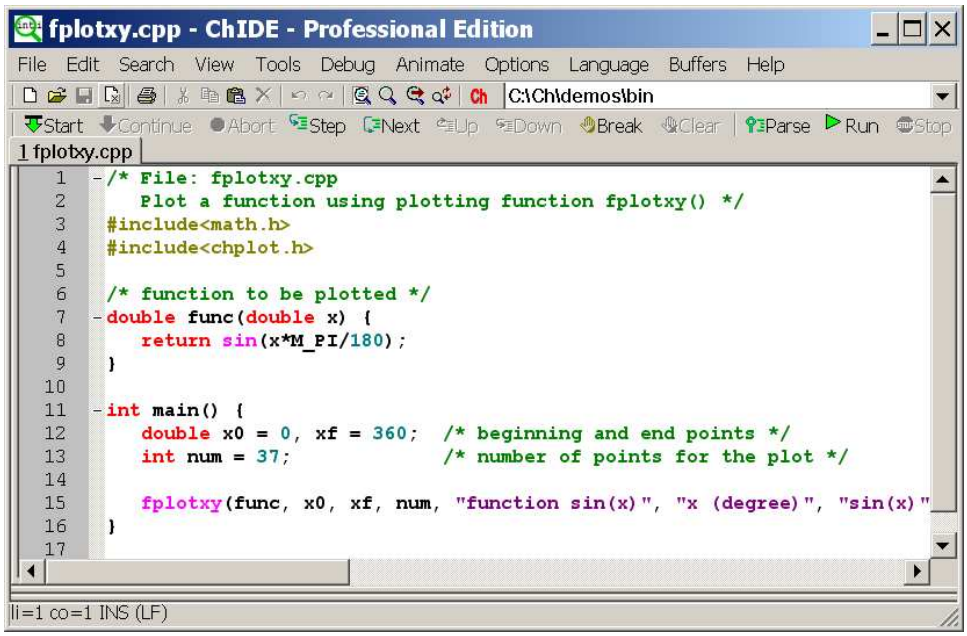


Figure 12: A program using the plotting function **fplotxy**().

If the modeless Command Line Arguments dialog is already visible, then the `'*'` is ignored.

The program in Figure 14 will accept the command line arguments and print them out. The command `Tools->Command Line Arguments` as shown in Figure 15 launches the modeless Command Line Argument dialog. Figure 16 shows how command line arguments are setup. The output from execution of this program with command line arguments is displayed in Figure 17.

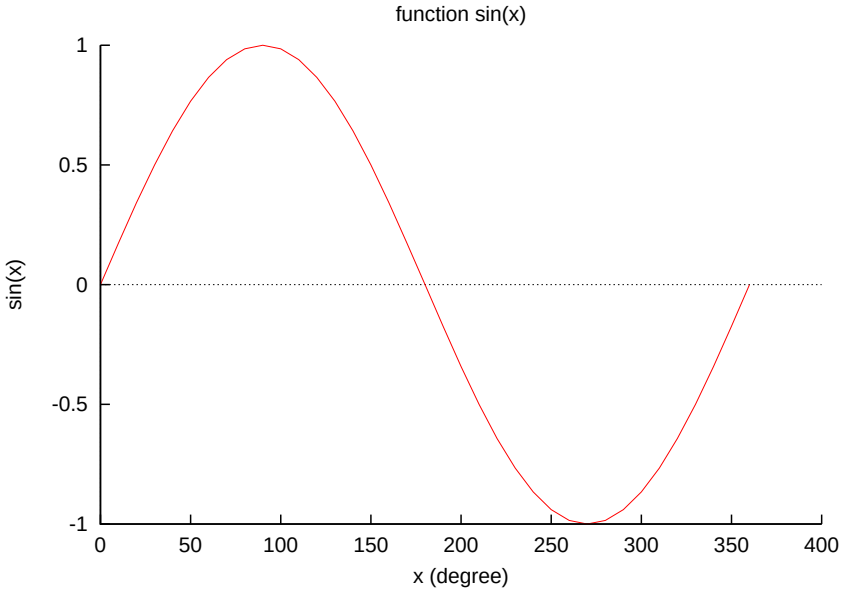


Figure 13: The output of the plotting program in Figure 12.

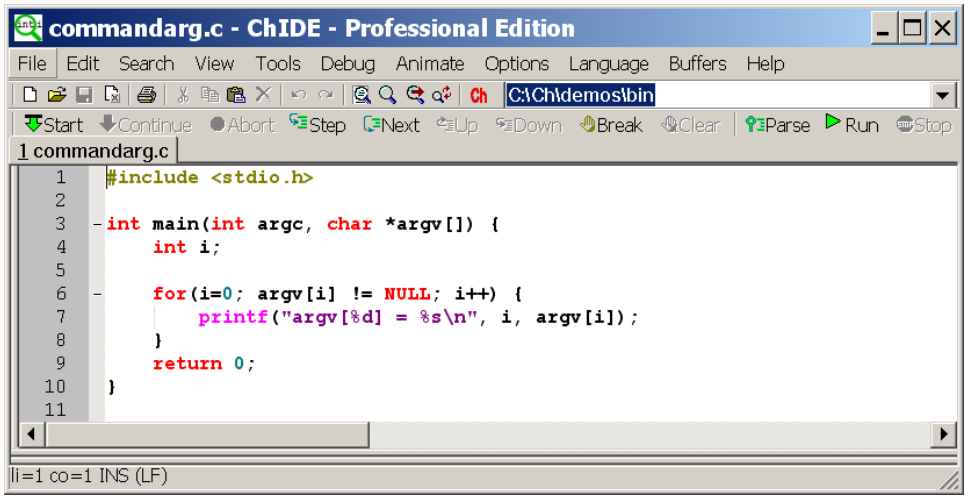


Figure 14: A program for handling command line arguments.

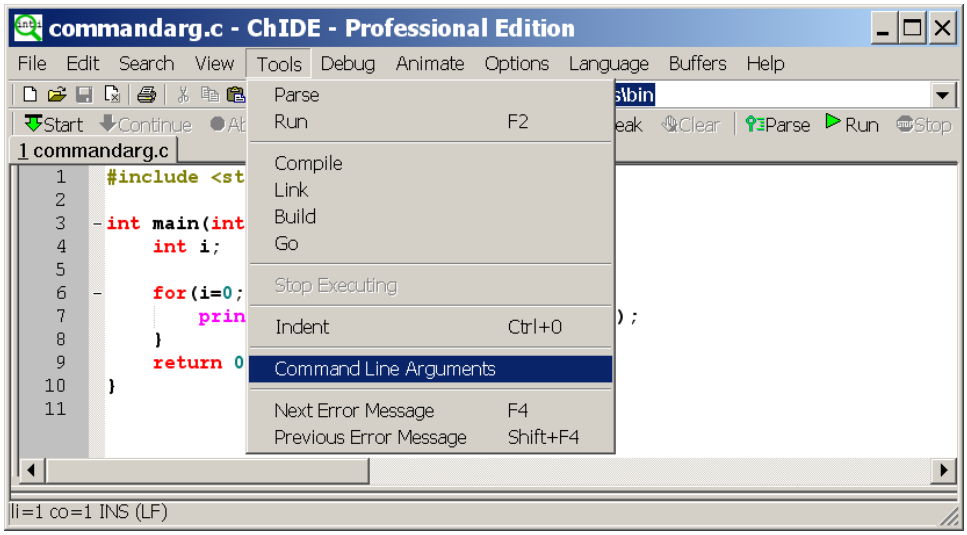


Figure 15: Launching the modal Command Line Arguments dialog.

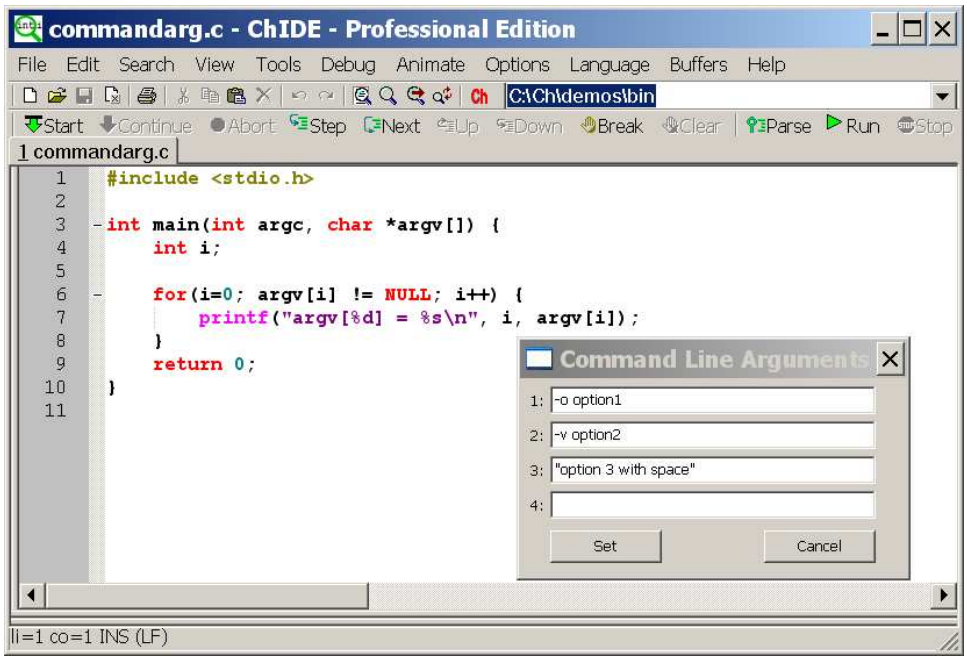


Figure 16: Setting command line arguments.

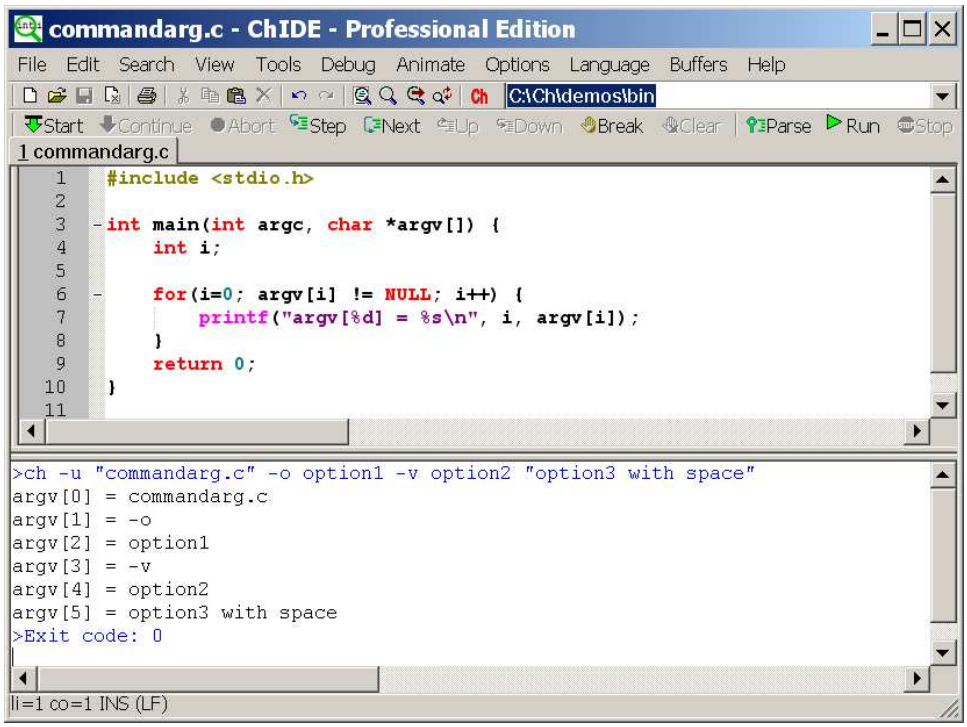


Figure 17: Executing the program with command line arguments.

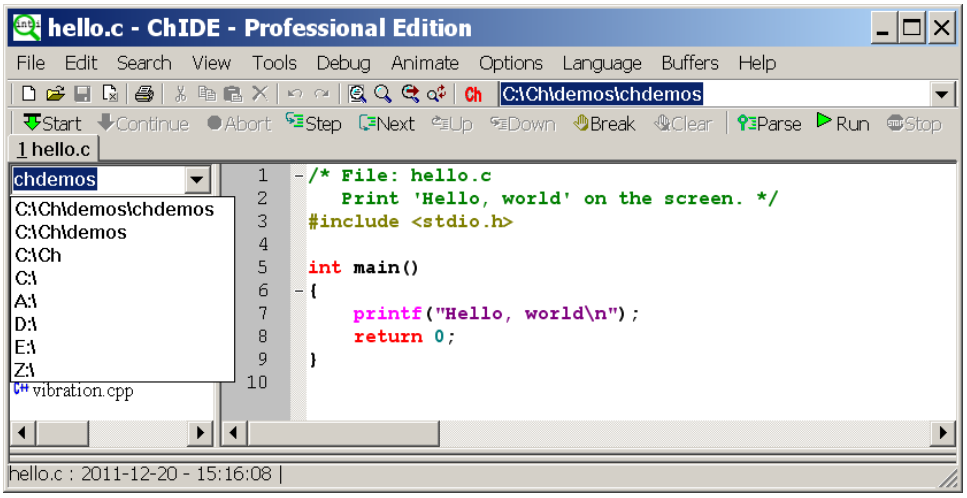


Figure 18: Displaying the parent directories of the current working directory.

2.6 Indenting C/Ch/C++ Programs

For readability and software maintenance, each line in a program should be properly indented. This is especially important for readability for a program with many nested loops and selection statements. The command `Tools->Indent` on the menu bar properly indents the program in the editing pane. You can also right click the file on the file name on the Tab bar, located below the debug bar, and then select the command `Indent` to indent the program. Figure 6 shows the command `Indent` when the file name `hello.c` on the Tab bar is right clicked.

3 Editing in ChIDE

Most text editing features in a word processor such as Microsoft Word or Notepad are available in ChIDE. Menus on the tool bar and menus under the command `Edit` on the menu bar can be used to edit programs in the editing pane. Some unique features for editing C/Ch/C++ programs in ChIDE are described in this section.

3.1 Browsing Files

Programs in the current working directory are displayed in the file browser pane, as shown in Figure 2. The current working directory is displayed at the end of the tool bar and also at the top of the file browser pane. The first entry under `Directories and Files` in the file browser pane is the parent directory of the current working directory. Clicking the arrow at the end of the current working directory in the file browser pane, all parent directories of the current working directory are displayed as shown in Figure 18. Selecting a parent directory, its contents will be displayed in the file browser pane.

The history of current working directories can be displayed by clicking the arrow at the end of the tool bar as shown in Figure 19. Selecting a directory in the history of the current working directories, the selected directory wil become the current working directory and its contents will be displayed in the file browser pane.

3.2 Edit

In Windows, Right clicking on the editing pane will also bring up the commonly used editing commands as shown in Figure 20.

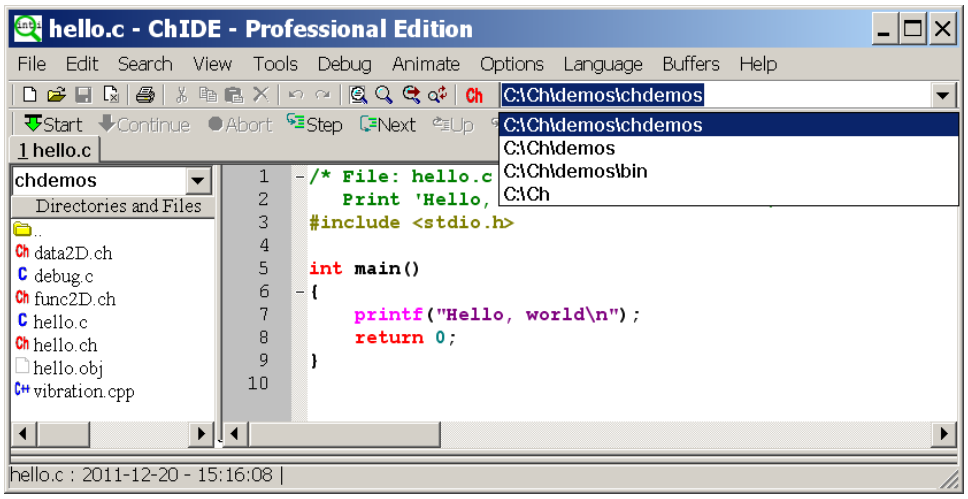


Figure 19: Displaying the history of current working directories.

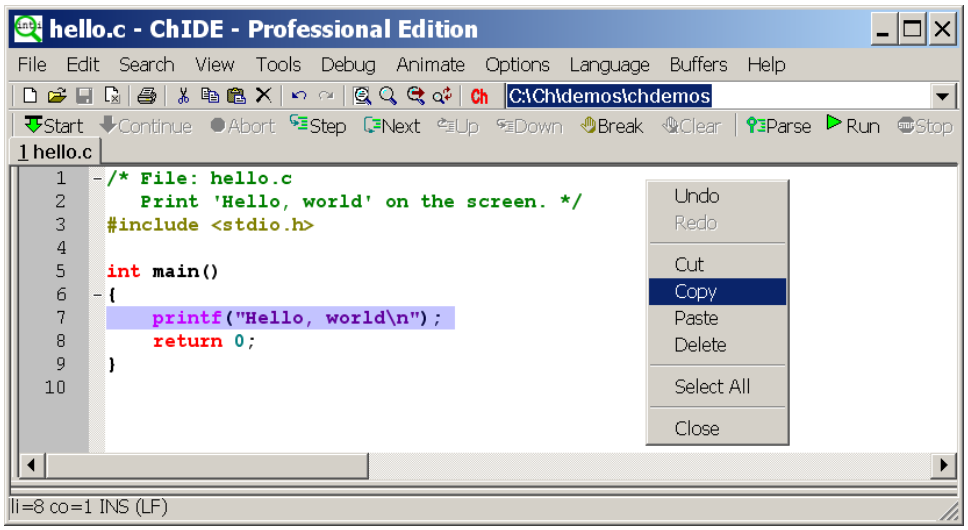


Figure 20: Using editing commands by right clicking on the editing pane.

As the user inputs the text into the editing pane, if the input string matches a word in the edited file, the matched word will be displayed. The user can hit the `Enter` key to automatically complete the input for the matched word. However, the user can type `Ctrl+Enter` to list all matched words, use the arrow key to select a word, then type `Enter` key to complete the word.

Rectangular regions of text can be selected in ChIDE by holding down the Alt key on Windows or the Ctrl key on Linux and Mac OS X while dragging the mouse over the text.

Key commands and abbreviations can be used to speed up editing. Table 2 in section 3.6 lists many key commands for quick editing. Abbreviations are described in section 3.7

3.3 Find and Replace

ChIDE has options to allow searching for words, regular expressions, matching case, in the reverse direction, wrapping around the end of the document. C style backslash escapes may be used to search and replace control characters. Replacements can be made individually, over the current selection or over the whole file. When regular expressions are used, tagged subexpressions can be used in the replacement text. Regular expressions will not match across a line end. For the user's convenience, the function key F3 can be used to search the next word.

Table 1: Commonly used commands and their corresponding keyboard commands in ChIDE.

Command	**Keyboard Command**
Help	F1
Run C/Ch/C++ program in Ch	F2
Find Next	F3
Find Previous	Shift+F3
Next Error Message	F4
Previous Error Message	Shift+F4
Start (Debug the program)	F5
Step (Single step)	F6
Next (Step over the next statement)	F7
Open or close Input/Output, Debug, Debug Command Pane	F8
Clear Input/Output Pane	F9
Animate a QuickAnimation file	F10
Send output for QuickAnimation	F11
Full screen	F12
Open or close File Browser Pane	Ctrl+1
Open or close Input/Output Pane	Ctrl+2
Open or close Debug Pane	Ctrl+3
Open or close Debug Command Pane	Ctrl+3

3.4 Changing Font Size

For the classroom presentation, the font size of the displayed program can be enlarged by clicking the command `View->Change Font Size`, and then make changes. In addition, the keyboard commands `Ctrl+Keypad+`, `Ctrl+Keypad-`, and `Ctrl+Keypad/` can be conveniently used during a presentation to magnify the font size, reduce the font size, and restore the font size to normal, respectively, as shown in Table 2 in section 3.6. Note that for a laptop without a separate Keypad, to use the keyboard commands, you need to turn on "Num Lock" by pressing `Shift+NumLk` key first. Then, use the keys on the keypad. For example, press the key `Ctrl+Keypad+` with the key for `'+'` next to the `Shift` key.

3.5 Folding

ChIDE supports folding for C/Ch/C++ and several other languages as presented in section 9. Fold points are based upon indentation for C/Ch/C++ and on counting braces for the other languages. The fold point markers can be clicked to expand or contract folds as shown in Figures 3 and 4 in section 2.2. The keyboard command `Ctrl+Shift+Click` in the fold margin will expand or contract all the top level folds. The command `Ctrl+Click` on a fold point to toggle it and perform the same operation on all children. The command `Shift+Click` on a fold point to show all children.

3.6 Keyboard Commands

Keyboard commands in ChIDE mostly follow common Windows and GTK+ conventions. All move keys (arrows, page up/down, home and end) allow to extend or reduce the stream selection when holding the Shift key, and the rectangular selection when holding the Shift and Alt keys. Some keys may not be available with some national keyboards or because they are taken by the system such as by a window manager on GTK+. Keyboard equivalents of menu commands are listed in the menus.

Table 1 lists the most commonly used commands and their corresponding keyboard commands.

Table 2 lists less commonly used commands with no menu equivalent.

By default, function keys F9, F10, F11, and F12 in Mac OS X are pre-binded to certain features. To use these function keys for ChIDE as shown in Table 1, you can disable these pre-binding with the following steps:

- Click the Apple symbol on the upper left corner.
- Click System Preferences.
- Click Keyboard & Mouse.
- Click Keyboard Shortcuts.
- Click to disable the pre-selected bindings for F9, F10, F11, and F12.

3.7 Abbreviations

Abbreviations in ChIDE can replace a short name with a predefined text for quick editing text or programs. To use an abbreviation, type it and use the `Edit->Expand Abbreviation` command or the `Ctrl+B` key to insert the expansion. The abbreviation is replaced by an expansion defined in the abbreviation files, one is global and the other is the user specific. The global abbreviations for writing C/Ch/C++ can be opened by the command

```
Options->Open ChIDE Global Abbreviation File
```

The global abbreviations can be overwritten by the user abbreviation. The user abbreviation file can be opened by the command

```
Options->Open ChIDE User Abbreviation File
```

An abbreviation file contains a list of entries of the form

```
abbreviation=expansion
```

An abbreviation name can have any character (except control characters such as CR and LF), including accented characters and multibyte characters for Asian languages such as Chinese.

The abbreviation names have properties files limits: they cannot start with sharp (#) or space or tab (but can have spaces inside); and they cannot have `'='` character inside. An abbreviation name is limited to 32 characters, which should be more than enough for an *abbreviation*.

An expansion may contain new line characters indicated by `'\n'`. The character `'|'` in an expansion marks the position where the caret will be after expansion. To include a literal `'|'` in an expasion, use `'||'`.

When expanding, the names don't need to be separated from the previous text, i.e. if you define $\ddot{e}$ as `'&eacute'`, you can expand it inside a word.

If a name is the ending of another one, only the shorter one will be expanded, i.e. if you define `'ring'` and `'gathering'`, the later will see only the `'ring'` part expanded.

The global programming abbreviations can be used to speed up the typing and indenting programs. Table 3 lists the global abbreviations predefined for writing C/Ch/C++ programs.

A sample abbreviation `hw` is included in the distributed default user abbreviation file. If you type the abbreviation `hw` followed by the `Ctrl+B` key, the contents for the header for a homework assignment, as shown in Figure 21, will be added in the editing pane conveniently. You may edit the user abbreviation file by the command

```
Options->Open ChIDE User Abbreviation File
```

to configure the abbreviation `hw` with your name and relevant information for a class or project.

Table 2: Less common commands and their corresponding keyboard commands in ChIDE.

Description	**Keyboard Command**
Magnify font size	Ctrl+Keypad+
Reduce font size	Ctrl+Keypad-
Restore font size to normal	Ctrl+Keypad/
Cycle through the opened files in the buffers	Ctrl+Tab
Indent block	Tab
Dedent block	Shift+Tab
Delete to start of word	Ctrl+BackSpace
Delete to end of word	Ctrl+Delete
Delete to start of line	Ctrl+Shift+BackSpace
Delete to end of line	Ctrl+Shift+Delete
Go to start of document	Ctrl+Home
Extend selection to start of document	Ctrl+Shift+Home
Go to start of display line	Alt+Home
Extend selection to start of display line	Alt+Shift+Home
Go to end of document	Ctrl+End
Extend selection to end of document	Ctrl+Shift+End
Go to end of display line	Alt+End
Extend selection to end of display line	Alt+Shift+End
Expand or contract a fold point	Ctrl+Keypad*
Create or delete a bookmark	Ctrl+F2
Select to next bookmark	Alt+F2
Scroll up	Ctrl+Up
Scroll down	Ctrl+Down
Line cut	Ctrl+L
Line copy	Ctrl+Shift+T
Line delete	Ctrl+Shift+L
Line transpose with previous	Ctrl+T
Line duplicate	Ctrl+D
Find matching preprocessor conditional, skipping nested ones	Ctrl+K
Select to matching preprocessor conditional	Ctrl+Shift+K
Find matching preprocessor conditional backwards, skipping nested ones	Ctrl+J
Select to matching preprocessor conditional backwards	Ctrl+Shift+J
Previous paragraph. Shift extends selection	Ctrl+[
Next paragraph. Shift extends selection	Ctrl+]
Previous word. Shift extends selection	Ctrl+Left
Next word. Shift extends selection	Ctrl+Right
Previous word part. Shift extends selection	Ctrl+/
Next word part. Shift extends selection	Ctrl+\

Table 3: The default global abbreviations and their expansions. (Continued)

Abbreviation	**Expansion**
com	/* | */
inc	#include <|>
myinc	#include "|"
def	#define |
main	function main()
mainarg	function main() with arguments
if	if statement
elseif	else if statement
else	else statement
for	for loop
while	while loop
do	do-while loop
switch	switch statement
foreach	foreach loop
a	[|] for an array index
c	'|' for a character
s	"|" for a string
p	(|) for parentheses
pi	M_PI|
epsilon	FLT_EPSILON|
cond	| ? : for conditional operator
sizeof	sizeof(|)
struct	struct structure
union	union structure
enum	enum structure
class	class structure
stdlib.h	include stdlib.h
time.h	include time.h
assert.h	include assert.h
complex.h	include complex.h
ctype.h	include ctype.h
errno.h	include errno.h
fenv.h	include fenv.h
float.h	include float.h
inttypes.h	include inttypes.h
iso646.h	include iso646.h
limits.h	include limits.h
locale.h	include locale.h
math.h	include math.h

Table 3: (Continued)

Abbreviation	Expansion
setjmp.h	include setjmp.h
signal.h	include stdarg.h
stdarg.h	include stdarg.h
stdbool.h	include stdbool.h
stddef.h	include stddef.h
stdint.h	include stdint.h
stdio.h	include stdio.h
stdlib.h	include stdlib.h
string.h	include string.h
tgmath.h	include tgmath.h
time.h	include time.h
wchar.h	include wchar.h
wctype.h	include wctype.h
chdl.h	include chdl.h
chplot.h	include chplot.h
chshell.h	include chshell.h
numeric.h	include numeric.h
func	a function definition
prot	|(); for a function prototype
call	|(); for calling a function
printf	printf("|\n");
scanf	scanf("|", &);
sin	sin(|)
a standard C function name	call the standard C function

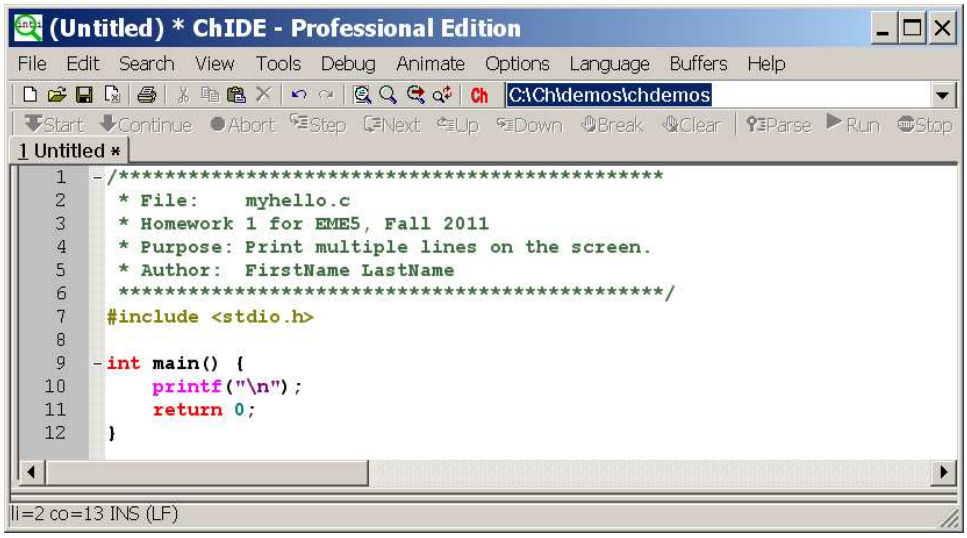


Figure 21: Using the abbreviation hw to create the header for a homework assignment.

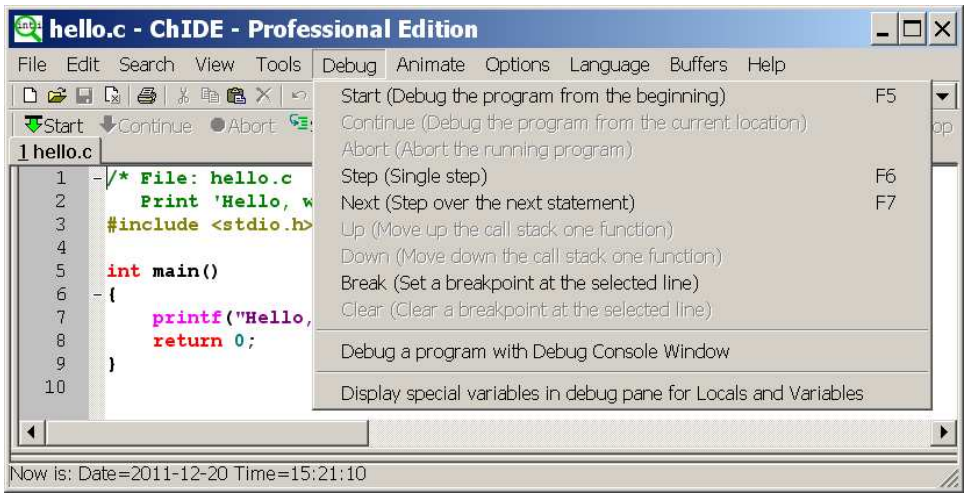


Figure 22: Debug menus.

3.8 Buffers

ChIDE has 20 buffers by default, each containing a file. The number of the default buffers can be changed in the user option file for ChIDE. The `Buffers` menu can be used to switch between buffers, either by selecting the file name or using the `Buffers->Previous File` and `Buffers->Next File` commands. The keyboard command `Ctrl+Tab` cycles through the opened files in the buffers as shown in Table 1 in section 3.6.

When all the buffers contain files, then opening a new file causes a buffer to be reused which may require a file to be saved. In this case an alert is displayed to ensure the user wants the file saved.

3.9 Sessions

A session is a list of file names and some options for ChIDE. You can save a complete set of your currently opened buffers as a session for fast batch-loading in the future. Sessions are stored as plain text files with the extension ".session".

Use the commands `File->Load Session` and `File->Save Session` to load/save sessions.

When ChIDE is closed, the opened buffers are saved in a session. When ChIDE is started next time, the previously saved session will be loaded automatically in the new session.

4 Debugging C/Ch/C++ Programs in ChIDE

The ChIDE has all capabilities available in a typical debugger for binary C programs. The debug interface commands, such as `Start` and `Step`, are available under the command `Debug` on the menu bar as shown in Figure 22. They are also available directly on the debug bar. The applicable commands on the debug bar at any point of debugging will be clickable. Non-clickable commands are dimmed.

4.1 Executing Programs in Debug Mode

The user can execute the program in the editing pane in the debug mode by the `Start` command or function key F5. The program will stop when a breakpoint is hit. The user can execute the program line by line either by command `Step` or `Next`. The command `Step` or function key F6 will step into a function whereas the command `Next` or function key F7 will step over the function to the next line. During debugging, the

command `Continue` can be invoked to continue the execution of the program till the program ends or it hits a breakpoint, which will be described in section 4.2.

If a program execution has failed and is taking too long to complete, then the command `Abort` can be used to stop the program. However, when a program is at the point of accepting input from the user such as by the input function **scanf()**, the program cannot be aborted only after the input function finishes.

4.2 Setting and Clearing Breakpoints

Before program execution or during the debugging of an executed program, new breakpoints can be added to stop the program execution when they are hit. A breakpoint for a line can be added by clicking the left margin of the line as shown in Figure 2. To clear the breakpoint, click the highlighted red mark on the left margin of the line. Breakpoints in the debugger can be examined by clicking `Breakpoints` on the debug pane selection bar above the debug pane as shown in Figure 2. The debug pane will display the breakpoint number and its location for each breakpoint. A breakpoint for the current line can also be added by clicking the command `Break`. on the debug bar It can also be deleted by clicking the command `Clear` on the debug bar. If no breakpoint has been set, the command `Clear` is non-clickable. A breakpoint cannot be set in a declaration statement; however, a breakpoint can be set for a declaration statement with initialization such as

```
int i = 10;
```

The program shall not be edited when it is being executed and debugged. Otherwise, a warning message

```
Warning:  Any changes made to the file during debugging will not
be reflected in the current debugging session
```

will be displayed. After a program is finished its execution, it can be edited. When a program is edited by deleting or adding new code, the breakpoints set for the program will be updated automatically.

Using debug commands inside the debug command pane, which will be described in section 4.5, a breakpoint can also be set for functions and controlling variables,

4.3 Monitoring Local Variables and Their Values in the Debug Pane

The command `Step` on the debug bar or under the command `Debug` on the menu bar can be used to step into a function. If the function is not in one of files loaded in the buffer already, the file containing the function will be loaded. At the end of the execution of the program, the file loaded during the debugging will be removed from the buffer. However, if a breakpoint has been set in the loaded file, the file will be kept in the buffer when the execution of the program is finished.

When a program is executed line by line by commands `Step` or `Next`, names and their corresponding values of variables in the current stack can be examined in the debug pane by clicking menu `Locals` on the debug pane selection bar. When control of the program execution is inside a function, the command `Locals` displays the values of local variables and arguments of the function. When control of the program execution is not in a function of a script, command `Locals` displays the values of global variables of the program. As shown in Figure 23, when program `func.c`, available in the directory `CHHOME/demos/bin`, is executed at line 9, highlighted by the color green, local integer variables `i` and `n` are 1 and 10, whereas the array `a` of double type contains 1, 2, 3, 4, and 5, as shown in the debug pane.

4.4 Monitoring Variables in Different Stacks and Their Values in the Debug Pane

The user can change the function stack during debugging. It can go `Up` to its calling function or move `Down` to the called function so that the variables within its scope can be displayed in the debug pane or

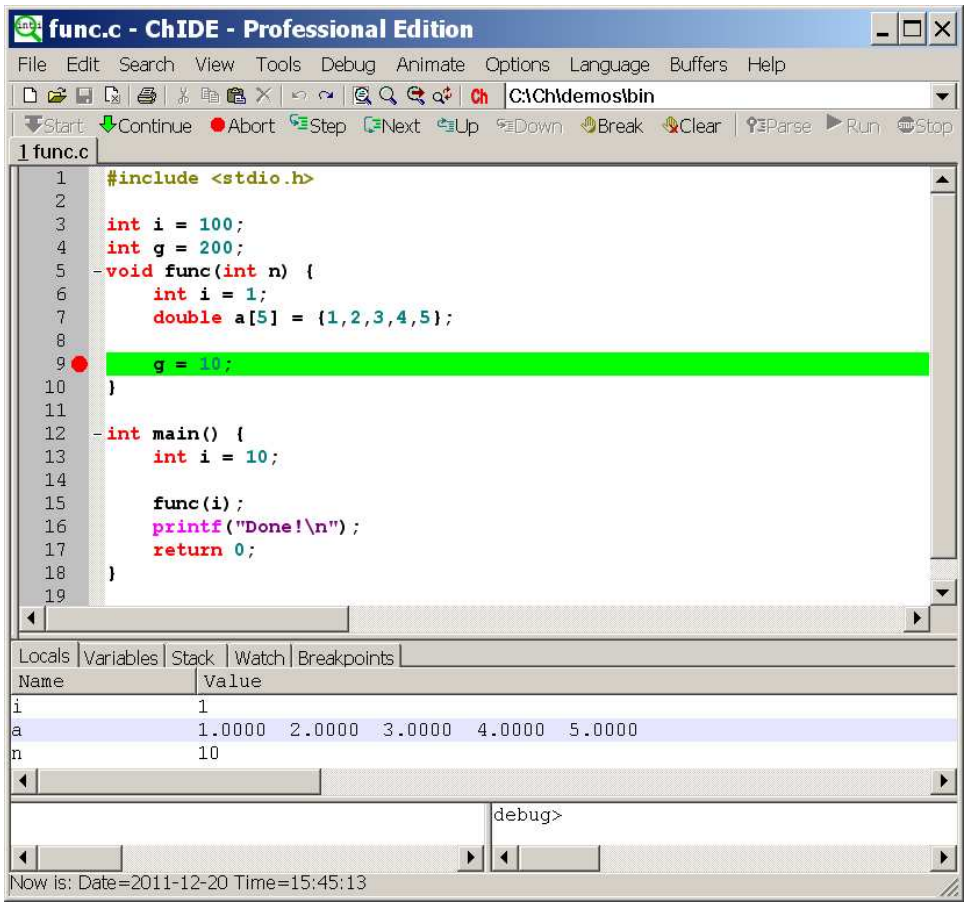


Figure 23: Displaying names and values of local variables in the currently called function.

accessed in the debug command pane. Different colors are used to highlight the current line and executing lines in the calling functions. For example, when clicking command `Up` in Figure 23, the control flow of the program moves to its calling function **main()** at line 15 as highlighted with the blue color in Figure 24. The menu `Down` as shown in Figure 23 is not clickable. But, the menu `Down` is clickable in Figure 24 when the current stack is moved up. The debug pane at this point displays the name and value of the variable `i`, the only regular variable, in the calling function **main().**

Command `Stack` displays function, member function, or program name and corresponding stack level in each stack. The current running function has stack level 0, whereas level n+1 is the function that has called a function with stack level n. For example, as shown in Figure 25, function `func()` is called by function **main()**, which in turn is invoked by the program `func.c`.

Names and their corresponding values of variables in all stacks can be displayed by the command `Variables` on the debug pane selection bar as shown in Figure 26. Stack levels are highlighted with the corresponding colors for the current line and executing lines in the calling functions in the editing pane as shown in Figure 24. In Figure 26, the program is stopped at line 9. Names and values of local variables inside functions `func()` and **main()** as well as global variables are displayed in the debug pane. As one can see, before line 9 is executed, the value of the global variable `g` is 200.

When the command

```
Display special variables in debug pane for Locals and Variables
```

in the debug menu shown in Figure 22 is clicked, names and values of special variables such as `__func__` will be displayed in the debug pane for commands `Locals` and `Variables`.

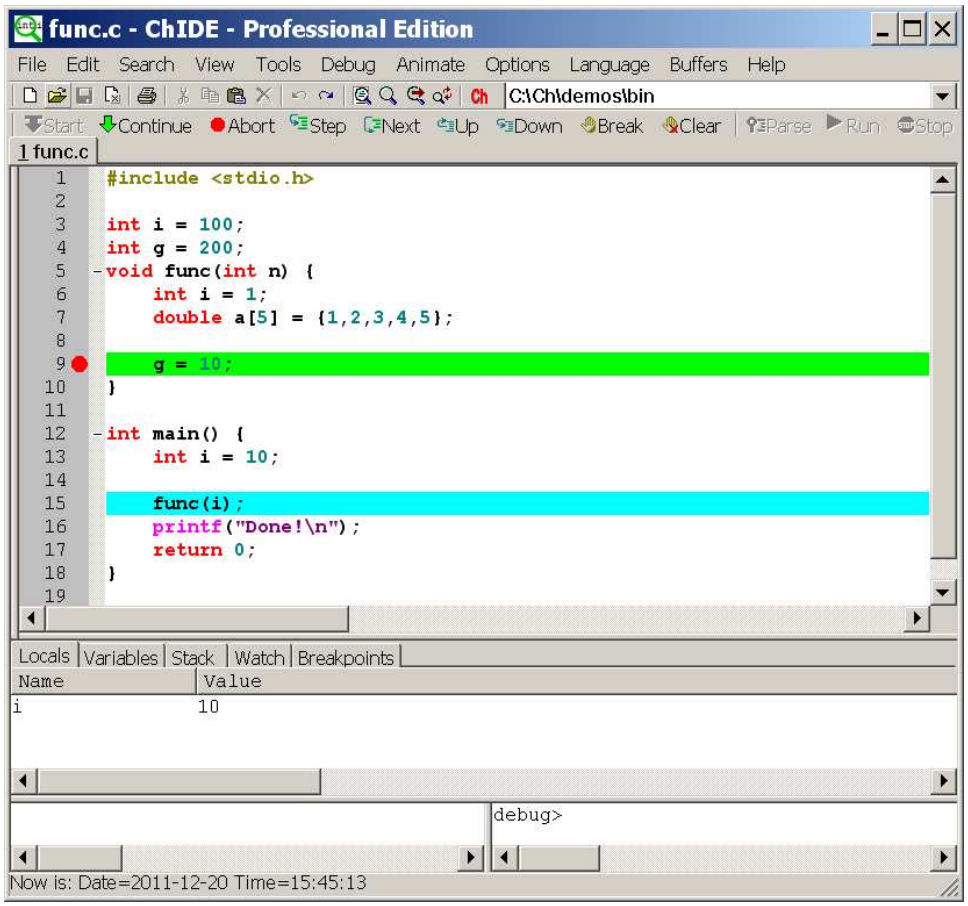


Figure 24: Displaying names and values of local variables in the calling function.

4.5 Using Debug Commands in the Debug Command Pane

Many debug commands inside the debug command pane are available during the debugging of a program. A prompt

```
debug>
```

inside the debug command pane indicates that the debugger is ready to accept debug commands. Type the command `help`, it will display all available commands as shown in Figure 27. The menu on the left before a colon shows a command and the description on the right explains the action taken for the command. All commands on the debug bar have corresponding commands in this interactive debug command pane. However, some features are available only through the debug command pane.

The variables, expressions, and functions can be manipulated by commands `assign, call`, and `print`. The command `assign` assigns a value to a variable, `call` invokes a function, and `print` prints out the value of a variable or expression including functions. It is invalid to print an expression of void type including a function with return type void. One can also just type an expression, the value of the expression will be displayed. If the expression is a function with the returning type of void, only the function is called. For example, commands

```
debug> assign i=2*10
debug> call func()
debug> print i
20
```

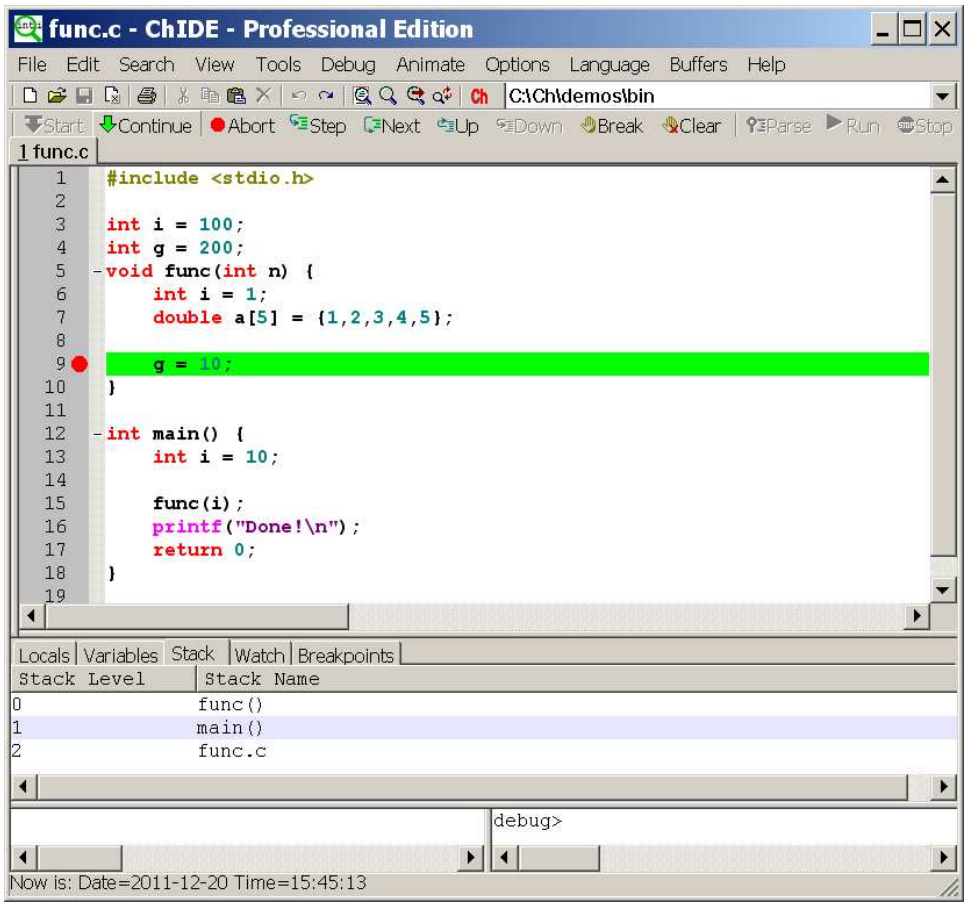


Figure 25: Displaying different stacks at the executing point.

```
debug> 2*i
40
debug>
```

assign the variable `i` with the value of 10, call function `func()`, and print out the value of the expression `2*i` when the variable `i` is valid in its current scope. As another example, when program `func.c` is executed and stopped at line 9 shown in Figure 28, the values of variables `a` and `i` as well as the expression `2*g` can be obtained by typing corresponding commands in the debug command pane.

The command `start` begins debugging a program. The optional command line arguments for the command `start` and `run` are processed and passed to the arguments for the function **main()**. For example, to run program `C:\Ch\demos\bin\commandarg.c` shown in Figure 17, the debug command

```
debug> start -o option1 -v option2 "option3 with space"
```

will assign the strings `"C:\Ch\demos\bin\commandarg.c"`, `"-0"`, `"option1"`, `"-v"`, `"option2"`, and `"option3 with space"` to elements `argv[0]`, `argv[1]`, `argv[2]`, `argv[3]`, `argv[4]`, and `argv[5]`, the argument `argv` of the main function

```
int main(int argc, char *argv[])
```

of the Ch script `commandarg.c`, respectively. The output on the Debug Console Window is similar to that displayed in the input/output pane in Figure 17. A command line argument with space should be enclosed within two double quotation marks as shown above for the string `"option3 with space"`.

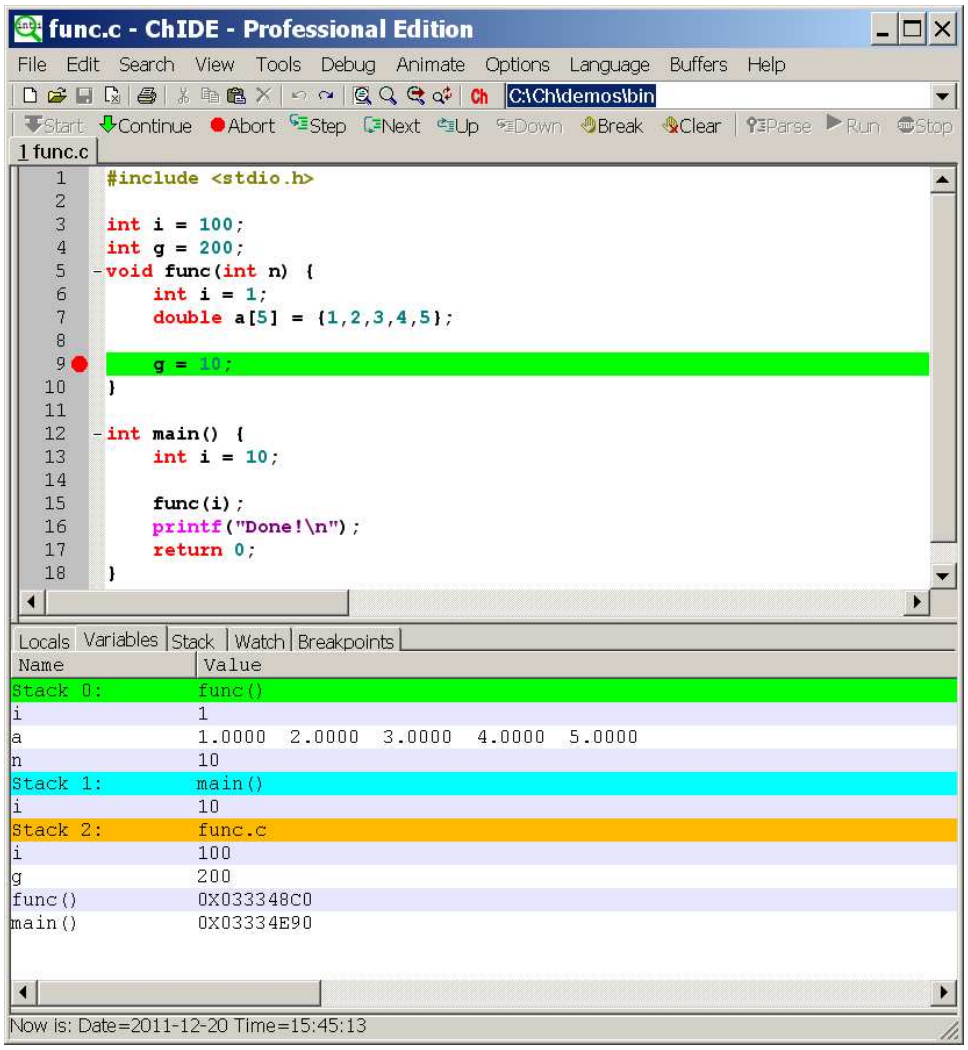


Figure 26: Displaying names and values of all variables in all stacks .

The program will stop when a breakpoint is hit. The command `run` will execute the program without debugging by ignoring breakpoints. Similar to commands on the debug bar, the user can execute the program line by line either by command `step` or `next`. The command `step` will step into a function whereas the command `next` will step over the function to the next line. During the debugging, the command `cont` can be invoked to continue the execution of the program till it hits a breakpoint or the program ends. The user can change the function stack during debugging. It can go up to its calling function or move down to the called function by the commands `up` and `down`, respectively, so that the variables within its scope can be accessed in the debug command pane. The function or program names in all stacks are displayed by the command `stack`. Names and their corresponding values of variables in the current stack are displayed by the command `locals`. Command `variables` displays names and values for all variables within its scope in each stack.

The command `watch` adds an expression, including a single variable, into a list of watched expressions. Watched expressions can be added before or during execution of a program. An expression can be removed from the list of the watched expressions by the `remove expr` command. The command `remove` removes all expressions in the watched list. For example, commands in the debug command pane

```
debug> watch 2*g
debug> watch i
```

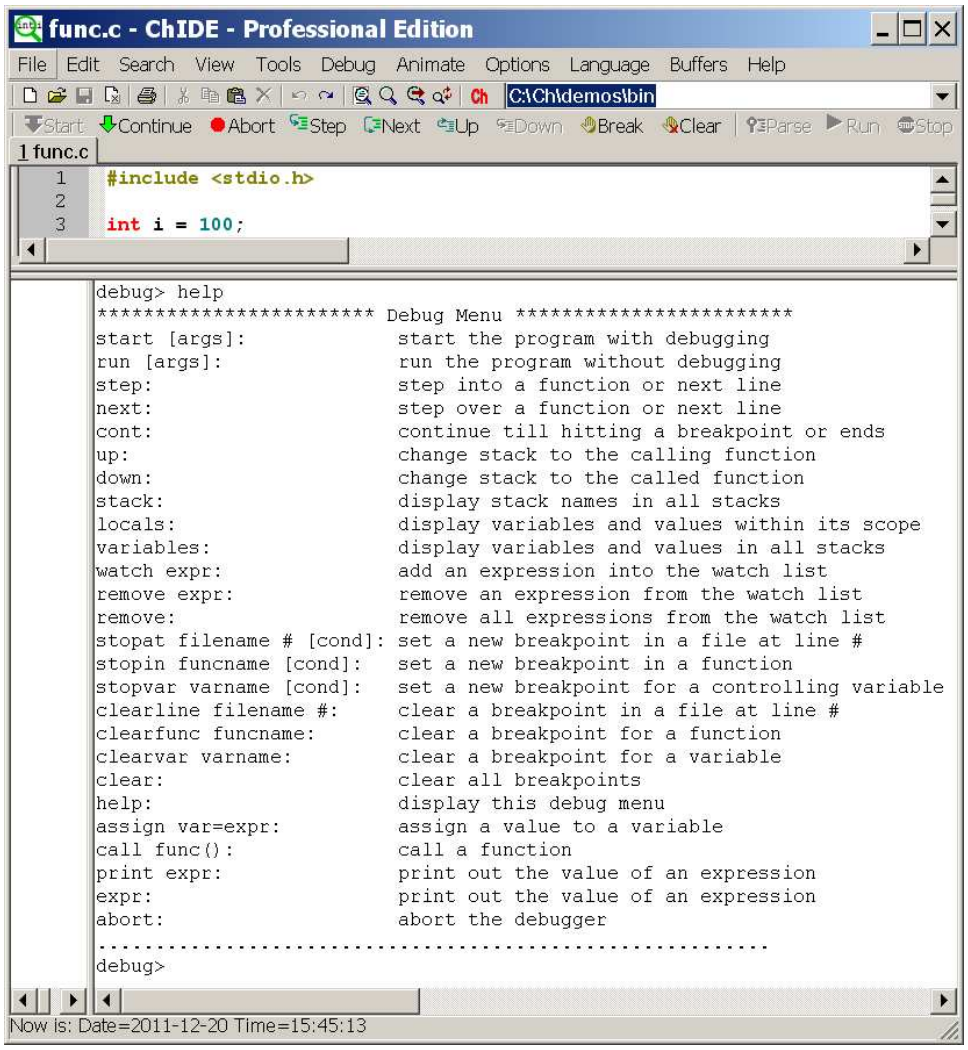


Figure 27: Debug commands in the debug command pane.

add expression `2*g` and variable i to a list of watched expressions as shown in Figure 29. When the program is stopped at a breakpoint or stepped into next statement, the values of these watched expressions can be viewed in the debug pane by clicking the command `Watch` on the debug pane selection bar as shown in Figure 29.

Before the program execution or during the debugging of an executed program, new breakpoints can be added to stop the program execution. A breakpoint can be setup based on three specifications: file name and line number, function, and controlling variable. When a breakpoint is setup in a function, the program will stop at its first executable line of the function. When a breakpoint is setup for a variable, the program will stop when the value of the variable changes. Each breakpoint can have an optional conditional expression. When a breakpoint location is reached, the conditional expression is evaluated if it exists. The breakpoint is hit only if the expression is either true or has changed which needs to be specified when the breakpoint was added. By default, the breakpoint is hit only if the expression is true. Command `stopat` sets a new breakpoint specified by a file name and line number in the subsequent arguments. The program breaks execution when it reaches this location. Command `stopin` sets a new breakpoint for a function. The program breaks execution when it reaches the first executable line of the function. Command `stopvar` sets a new breakpoint for a controlling variable. The variable is evaluated while the program is running. The program breaks execution when the value of the variable changes. When each of these command is invoked,

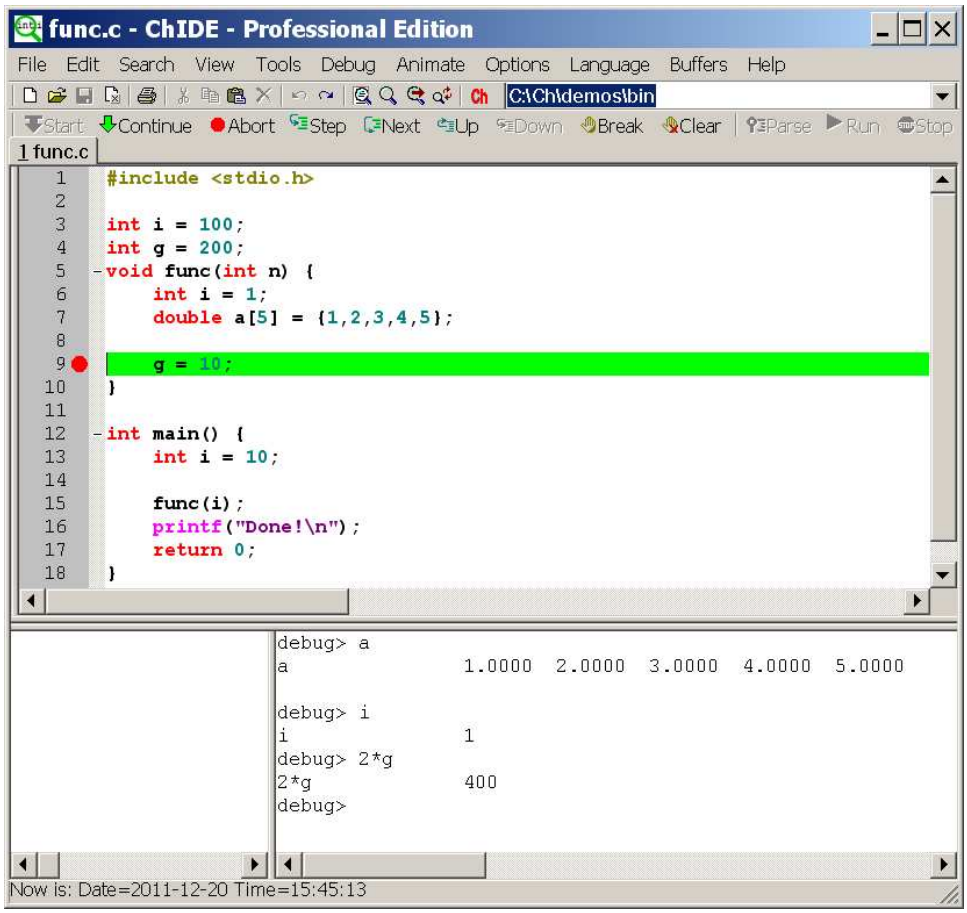


Figure 28: Using debug commands in the debug command pane.

a breakpoint is appended to the list of breakpoints. The optional conditional expression and triggering method for each breakpoint are passed as the last two arguments of these commands. For example, the syntaxes for setting a breakpoint in a file with a complete path and line number are as follows.

```
debug> stopat filename #
debug> stopat filename # condexpr
debug> stopat filename # condexpr condtrue
```

The symbol # should be substituted by a line number. When a breakpoint location is reached, the optional expression `condexpr` is evaluated. If the argument `condtrue` is true or missing, the breakpoint will be hit if the value for the expression is true; otherwise, the breakpoint will be hit if the value for the expression has changed. For example, the command

```
debug> stopat C:/Ch/demos/bin/func.c 6
```

sets a breakpoint in file `func.c` located at the directory C:/Ch/demos/bin at line 6. The command

```
debug> stopat C:/Ch/demos/bin/func.c 6 i+j 1
```

sets a breakpoint in file `func.c` at line 6. When the breakpoint location in file func.c at line 6 is reached, the expression `i+j` is evaluated and the breakpoint will be hit if the value for the expression `i+j` is true. The above command is the same as

```
debug> stopat C:/Ch/demos/bin/func.c 6 i+j
```

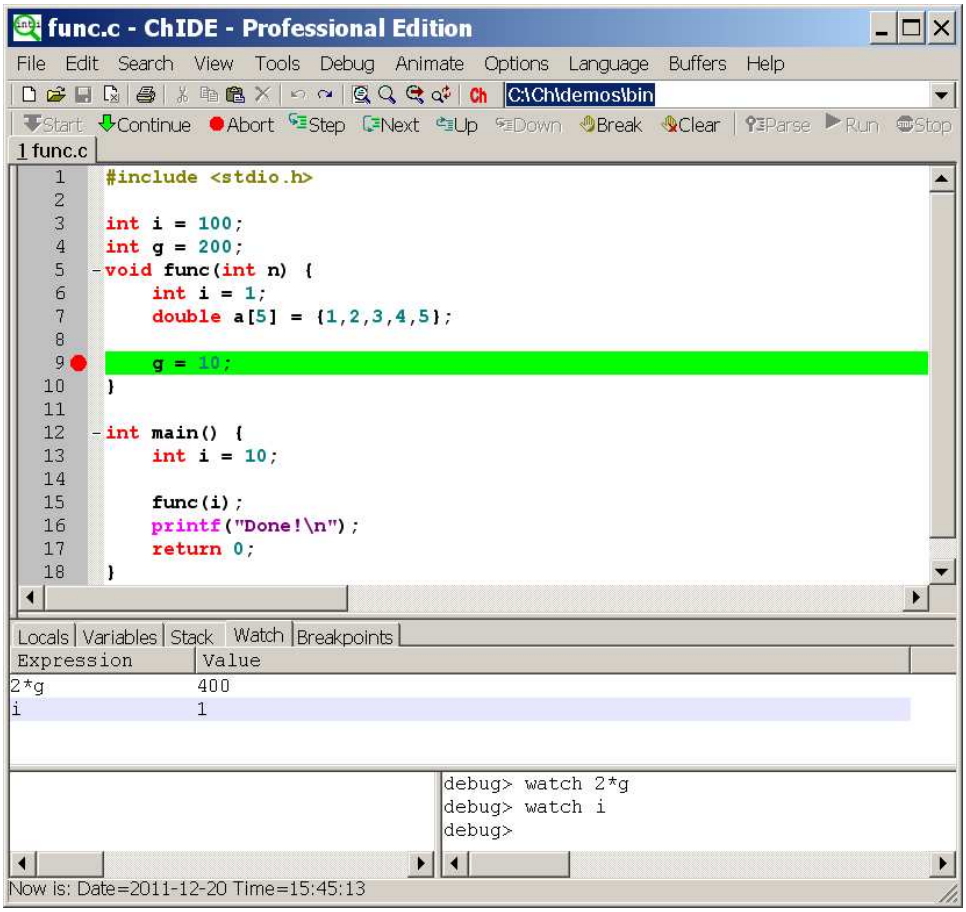


Figure 29: Setting watch expressions and variables inside the debug command pane to display their values in the debug pane.

The command

```
debug> stopat C:/Ch/demos/bin/func.c 6 i+j 0
```

sets a breakpoint in file `func.c` at line 6. When the breakpoint location in file `func.c` at line 6 is reached, the expression `i+j` is evaluated and the breakpoint will be hit if the value for the expression `i+j` has changed. On the other hand, commands `clearline`, `clearfunc`, and `clearvar` with proper arguments remove a breakpoint of line, function, and variable type in the list, respectively. Command `clear` removes all breakpoints in the debugger.

If a program execution has failed and is taking too long to complete, then the command `abort` can be used to stop the program.

The debug command pane can be cleared by clicking the command `View->Clear Debug Command Pane` as shown in Figure 10.

4.6 Using the Debug Console Window for Input and Output

Normally, the standard input and output of a program are handled in the input/output pane. For some applications, such as Windows console applications using the function hitkb(), the user may interface with the program through a console windows in debug mode. The debug console window can be opened and closed by the command `View->Debug Console Window`. When a program is executed in the debug mode, the standard input, output, and error streams are then redirected in a separate Debug Console Window shown in Figure 30.

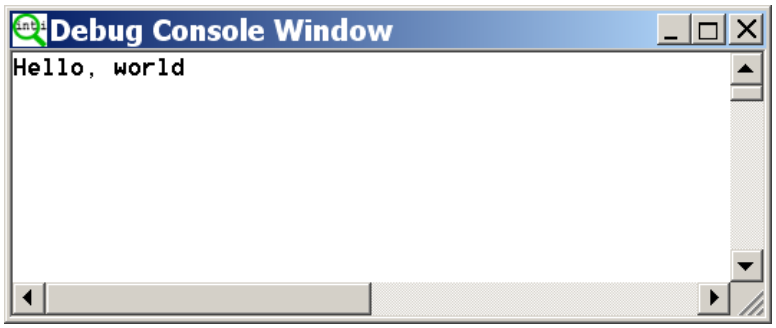


Figure 30: The Debug Console Window for input/output in debugging.

Figure 31: A Ch icon on a desktop in Windows, Linux, and Mac OS X.

By default, the console window always stays on the top of other windows. This default behavior can be turned off or on by the command `View->Debug Console Window Always on Top`. The contents of the debug console window can be cleared by the command `Debug->Clear Debug Console Window` as shown in Figure 10. The colors for background and text as well as the windows size and font size of the debug console window can be changed by right clicking the ChIDE icon on the upper left corner of the window and selecting the menu `Properties` to make changes. Note that for Windows Vista, you need to run ChIDE with the administrative privilege to make such a change.

5 Getting Started with Ch Command Shell

Ch can be used as a command shell in which commands are processed. Like other commonly used shells such as the MS-DOS shell, Bash-shell, or C-shell, commands can be executed in a Ch shell. Unlike these conventional shells, expressions, statements, functions and programs in C and C++ can be readily executed in a Ch shell.

A Ch shell can be launched by running the command **ch**. In Windows, Linux, and Mac OS X, a Ch command shell can also be conveniently launched by clicking the red-colored **Ch** icon, shown in Figure 31, on the desktop or on the tool bar of the ChIDE.

Assume the user account is the administrator, after a Ch shell is launched in Windows, by default, the screen prompt of the shell window becomes

```
C:/Documents and Settings/Administrator>
```

where `C:/Documents and Settings/Administrator` is the user's *home directory* on the desktop as shown in Figure 32. The colors of the text and background as well as the window size and font size of the shell window can be changed by right clicking the Ch icon at the upper left corner of the window, and selecting the menu `Properties` to make changes. Note that for Windows Vista, you need to run ChIDE with the administrative privilege to make such a change. The displayed directory `C:/Documents and Settings/Administrator` is also called the *current working directory*. If the user account is not the administrator, the account name *Administrator* shall be changed to the appropriate user account name. The prompt indicates that the system is in a Ch shell and is ready to accept the user's terminal keyboard input. The default prompt in a Ch shell can be reconfigured. If the input typed in is syntactically correct, it will be executed successfully. Upon completion of the execution, the system prompt > will appear again. If an

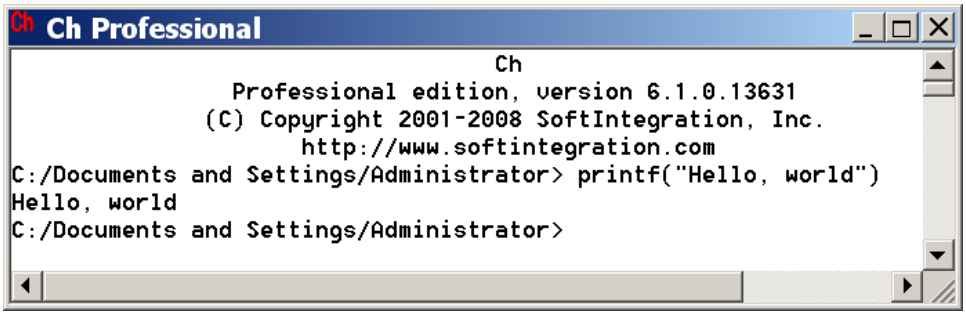


Figure 32: A Ch command shell.

Table 4: Portable commands for handling files.

Command	Usage	Description
cd	cd	change to the home directory
	cd *dir*	change to the directory *dir*
cp	cp *file1 file2*	copy *file1* to *file2*
ls	ls	list contents in the working directory
mkdir	mkdir *dir*	create a new directory *dir*
pwd	pwd	print (display) the name of the working directory
rm	rm *file*	remove *file*
chmod	chmod +x *file*	change the mode of *file* to make it executable
chide	chide *file.c*	launch ChIDE for editing and executing *file.c*

error occurs during the execution of the program or expression, the Ch shell prints out the corresponding error messages to assist the user in debugging the program.

All statements and expressions of C can be executed interactively in a Ch command shell. For example, the output `Hello, world` can be obtained by calling the function **printf**() interactively as shown below and as seen in Figure 32.

```
C:/Documents and Settings/Administrator> printf("Hello, world")
Hello, world
```

In comparison with Figure 32, the last prompt `C:/Documents and Settings/Administrator>` is omitted to save the space in the presentation of this book. Note that the semicolon at the end of a statement in a C program is optional when the corresponding statement is executed in command mode. There is no semicolon in calling the function **printf**() in the above execution.

5.1 Portable Commands for Handling Files

At the system prompt >, not only C programs and statements, but also any other commands (such as **pwd** for printing the current working directory) can be executed. In this scenario, Ch is used as a command shell in the same manner as MS-DOS shell in Windows.

Commands can be executed in a Ch command shell or in a Ch program. There are hundreds of commands along with their respective online documentation in the system. No one knows all of them. Every computer wizard has a small set of working tools that are used all the time, plus a vague idea of what else is out there. In this section, we will describe how to use the most commonly used commands, listed in Table 4, for handling files through examples. It should be emphasized again that these commands running in the Ch shell are portable across different platforms such as Windows, Linux, or Mac OS X. Using these commands, a user can effectively manipulate files on the system to run C programs.

Assume that Ch is installed in `C:/Ch` in Windows, the default installation directory. The current working directory is `C:/Documents and Settings/Administrator`, which is also the user's home directory. The application of portable commands for file handling can be illustrated by interactive execution of commands in a Ch shell as shown below.

```
C:/Documents and Settings/Administrator> mkdir c99
C:/Documents and Settings/Administrator> cd c99
C:/Documents and Settings/Administrator/c99> pwd
C:/Documents and Settings/Administrator/c99
C:/Documents and Settings/Administrator/c99> cp C:/Ch/demos/bin/hello.c hello.c
C:/Documents and Settings/Administrator/c99> ls
hello.c
C:/Documents and Settings/Administrator/c99> chide hello.c
```

As shown in **Usage** in Table 4, the command **mkdir** takes one argument as a directory to be created. We first create a directory called `c99` using the command

```
mkdir c99
```

Then, we change to this new directory `C:/Documents and Settings/Administrator/c99` using command

```
cd c99
```

Next, we display the current working directory with the command

```
pwd
```

A C program `hello.c` shown in Figure 3 in the directory `C:/Ch/demos/bin` is copied to the working directory with the same file name using the command

```
cp C:/Ch/demos/bin/hello.c hello.c
```

Files in the current directory are listed using the command

```
ls
```

At this point, there is only one file `hello.c` in the directory `C:/Documents and Settings/Administrator/c99`. It is recommended that you save all your developed C programs in this directory so that you may easily find all programs later on. Finally, program `hello.c` is launched by the command

```
chide hello.c
```

to be edited and executed in ChIDE as shown in Figure 3. For a classroom presentation, sometimes, it is more convenient to open multiple source files by a single command as shown below:

```
> chide file1.c file2.c header.h
```

To use a command dealing with a path with white space, the path needs to be placed inside a pair of double quotation marks, as shown below, to remove file `hello.c`.

```
> rm "C:/Documents and Settings/Administrator/c99/hello.c"
```

5.2 Setup Search Paths for Commands, Header Files, and Function Files in Ch

When a command is typed into a prompt of a command shell for execution, the command shell will search for the command in prespecified directories. In a Ch shell, the system variable **_path** of string type contains the directories to be searched for the command. Each directory is separated by a semicolon inside the string **_path**. When a Ch command shell is launched, the system variable **_path** contains some default search paths. For example, in Windows, the default search paths are

```
C:/Ch/bin;C:/Ch/sbin;C:/Ch/toolkit/bin;C:/Ch/toolkit/sbin;C:/WINDOWS;C:/WINDOWS/SYSTEM32;
```

The user can add new directories to the search paths for the command shell by using the string function **stradd()** in the startup file, which will be discussed in detail a little later. This function adds arguments of string type and returns it as a new string. For example, the directory `C:/Documents and Settings/Administrator/c99` is not in the search paths for a command. If you try to run program `hello.c` in this directory when the current working directory is `C:/Documents and Settings/Administrator`, the Ch shell will not be able to find this program, as shown below, and give two error messages.

```
C:/Documents and Settings/Administrator> hello.c
ERROR: variable 'hello.c' not defined
ERROR: command 'hello.c' not found
```

When Ch is launched or a Ch program is executed, by default, it will execute the startup file **.chrc** in Unix such as Linux and Mac OS X or **_chrc** in Windows in the user's home directory if the startup file exists. In the remaining presentation, it is assumed that Ch is used in Windows with a startup file **_chrc** in the user's home directory. This startup file typically sets up the search paths for commands, header files, function files, etc. In Windows, a startup file **_chrc** with default setup is created in the user's home directory during installation of Ch. However, there is no startup file in a user's home directory in Unix by default. The system administrator may add such a startup file in a user's home directory. However, the user can execute Ch with the option `-d` as follows

```
ch -d
```

to copy a sample startup file from the directory `CHHOME/config/` to the user's home directory if there is no startup file in the home directory yet. Note that `CHHOME` is not the string `"CHHOME"`, instead it uses the file system path under which Ch is installed. For example, by default, Ch is installed in `C:/Ch` in Windows and `/usr/local/ch` in Unix. In Windows, the command in a Ch shell below

```
C:/Documents and Settings/Administrator> ch -d
```

will create a startup file **_chrc** in the user's home directory `C:/Documents and Settings/Administrator`. This local Ch initialization startup file **_chrc** can be opened by the following command on the menu bar

```
Options->Open Ch Local Startup File
```

to edit the search paths in ChIDE, as shown in Figure 33. In Linux, the above command `ch -d` will also create an icon for Ch on the desktop. If Ch is installed with a ChIDE, an icon for ChIDE will also be created on the desktop.

To include the directory `C:/Documents and Settings/Administrator/c99` in the search paths for a command, the following statement

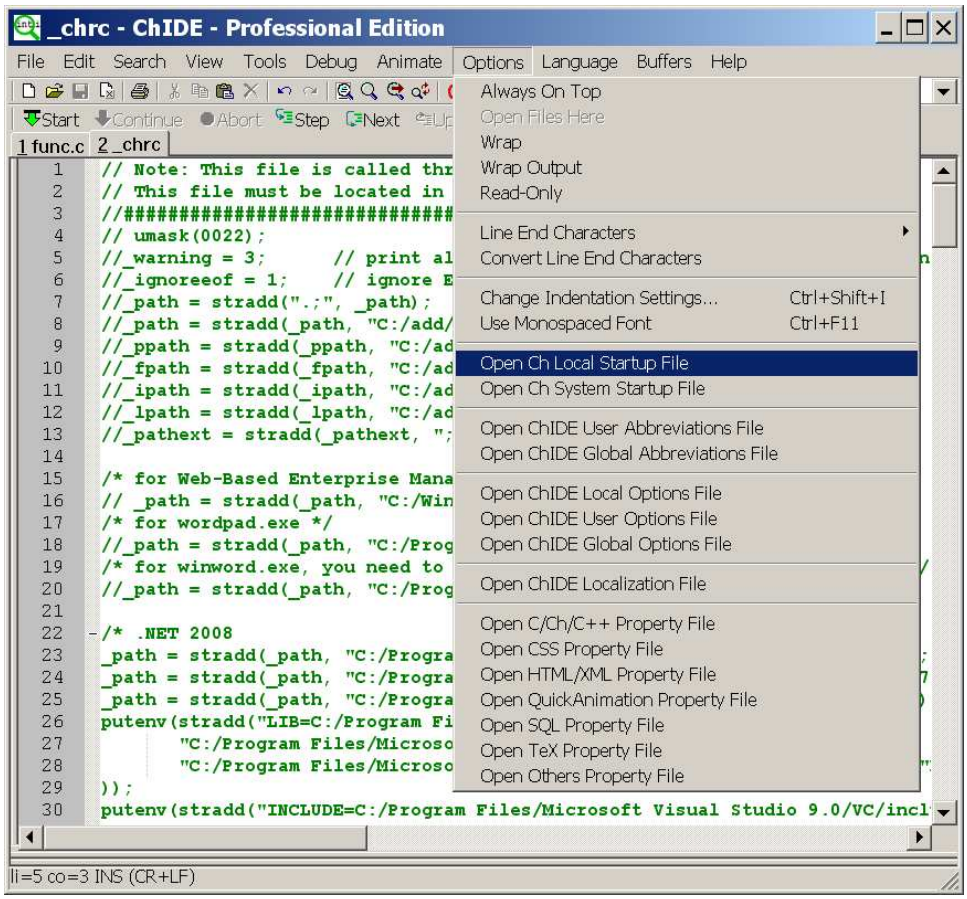


Figure 33: Open the local Ch initialization startup file for editing.

```
_path = stradd(_path, "C:/Documents and Settings/Administrator/c99;");
```

needs to be added to the startup file **_chrc** in the user's home directory so that the command `hello.c` in this directory can be invoked regardless of what the current working directory is. After the directory `C:/Documents and Settings/Administrator/c99` has been added to the search path, **_path**, you need to restart a Ch command shell. Then, you will be able to execute the program `hello.c` in this directory as shown below.

```
C:/Documents and Settings/Administrator> hello.c
Hello, world
```

In Unix such as Linux and Mac OS X, the search paths for commands by default do not contain the current working directory. To include the current working directory in the search paths for a command, the following statement

```
_path = stradd(_path, ".;");
```

needs to be added in startup file **.chrc** in the user's home directory. Function call `stradd(_path, ".;")` adds the current directory represented by '.' to the system search paths **_path**.

Similar to **_path** for commands, the header files in Ch are searched in directories specified in the system variable **_ipath**. Each path is also delimited by a semicolon. For example, the statement below

```
_ipath = stradd(_ipath, "C:/Documents and Setting/Administrator/c99;");
```

adds the directory `C:/Documents and Setting/Administrator/c99` to the search paths for header files included by the preprocessing directive **#include** such as

```
#include <headerfile.h>
```

One can also add this directory to the search paths **_fpath** for function files by the statement

```
_fpath = stradd(_fpath, "C:/Documents and Setting/Administrator/c99;");
```

A function file contains the function definition, which will be described in section 5.5.

5.3 Interactive Execution of C/Ch/C++ Programs

It is very simple and easy to run C programs interactively without compilation in a Ch shell. For example, assume that `C:/Documents and Settings/Administrator/c99` is the current working directory as presented in section 5.1. The program `hello.c` in this directory can be executed in Ch to get the output of `Hello, world` as shown below.

```
C:/Documents and Settings/Administrator/c99> hello.c
Hello, world
C:/Documents and Settings/Administrator/c99> _status
0
```

The exit code from executing a program in a Ch command shell is kept in the system variable **_status**. Because the program `hello.c` has been executed successfully, the exit code is 0 as shown in the above output when **_status** is typed in the command line.

In Unix such as Linux and Mac OS X, in order to readily use the C program `hello.c` as a command, the file has to be executable. The command **chmod** can change the mode of a file. The following command

```
chmod +x hello.c
```

will make the program `hello.c` executable so that it can run in a Ch command shell.

5.4 Interactive Execution of C/Ch/C++ Expressions and Statements

For simplicity, only the prompt `>` in a Ch command shell will be displayed in the remaining presentation. If a C expression is typed in the command shell, it will be evaluated by Ch and the result then will be displayed on the screen. For example, if the expression `1+3*2` is typed in, the output will be 7 as shown below.

```
> 1+3*2
7
```

Any valid C expression can be evaluated in a Ch shell. Therefore, Ch can be conveniently used as a calculator.

As another example, one can declare a variable at the prompt and then use the variable in the subsequent calculations as shown below.

```
> int i
> sizeof(int)
4
> i = 30
```

```
30
> printf("%x", i)
1e
> printf("%b", i)
11110
> i = 0b11110
30
> i = 0x1E
30
> i = -2
-2
> printf("%b", i)
11111111111111111111111111111110
> printf("%32b", 2)
00000000000000000000000000000010
```

In the above C statements, variable `i` is declared as int type with 4 bytes. Then, the integer value 30 for `i` is displayed in decimal, hexadecimal, and binary numbers. The integral constants in different number systems can also be assigned to variable `i` as seen above. Finally, the two's complement representation of the negative number -2 is also displayed. Characteristics for all other data types in C can also be presented interactively.

By default, a value of float or double type is displayed with two or four digits after the decimal point, respectively. For example,

```
> float f = 10
> 2*f
20.00
> double d = 10
> d
10.0000
```

All C operators can be used interactively as shown below.

```
> int i=0b100, j = 0b1001
> i << 1
8
> printf("%b",  i|j)
1101
```

The concept of pointers and addresses of variables can be illustrated as shown below.

```
> int i=10, *p
> &i
1eddf0
> p = &i
1eddf0
> *p
10
> *p = 20
```

```
20
> i
20
```

In this example, the variable p of pointer to int points to the variable i. The working principle for pointer to pointer can also be interactively illustrated in the same manner. In the next example, the relation of arrays and pointers is illustrated as follows:

```
> int a[5] = {10,20,30,40,50}, *p;
> a
1eb438
> &a[0]
1eb438
> a[1]
20
> *(a+1)
20
> p = a+1
1eb43c
> *p
20
> p[0]
20
```

Expressions a[1], *(a+1), *p, and p[0] all refer to the same element. Multi-dimensional arrays can also be handled interactively. The boundary of an array is checked in Ch to detect potential bugs. For example,

```
> int a[5] = {10,20,30,40,50}
> a[-1]
WARNING: subscript value -1 less than lower limit 0
10
> a[5]
WARNING: subscript value 5 greater than upper limit 4
50
> char s[5]
> strcpy(s, "abc")
abc
> s
abc
> strcpy(s, "ABCDE")
ERROR: string length s1 is less than s2 in strcpy(s1,s2)
ABCD
> s
ABCD
```

The allowed indices for array a of 5 elements are from 0 to 4. Array s can only hold 5 characters including a null character. Ch can catch bugs in existing C code related to the array boundary overrun such as these.

The alignment of a C structure or C++ class can also be examined as shown below.

```
> struct tag {int i; double d;} s
> s.i =20
20
> s
.i = 20
.d = 0.0000
> sizeof(s)
16
```

In this example, although the sizes of int and double are 4 and 8, respectively, the size of structure `s` with two fields of int and double types is 16, instead of 12, for the proper alignment.

5.5 Interactive Execution of C/Ch/C++ Functions

A program can be divided into many separate files. Each file consists of many related functions, which can be accessible to any part of a program. All functions in the C standard libraries can be executed interactively and can be used inside user defined functions. For example, in the interactive execution:

```
> srand(time(NULL))
> rand()
4497
> rand()
11439
> double add(double a, double b) {double c; c=a+b+sin(1.5); return c;}
> double c
> c = add(10.0, 20)
30.9975
```

The random number generator function **rand()** is seeded with a time value in **srand(time(NULL)**. Function `add()` which calls type-generic mathematical function **sin()** is defined at the prompt and then used.

A file that contains more than one function definition is usually suffixed with `.ch` to identify itself as part of a Ch program. One can create a function file in a Ch programming environment. A *function file* in Ch is a file that contains only one function definition. The name of a function file ends in `.chf`, such as `addition.chf`. The names of the function file and function definition inside the function file must be the same. The functions defined using function files are treated as if they were system built-in functions in Ch.

Similar to **_path** for commands, a function is searched based on the search paths in the system variable **_fpath** for function files. Each path is delimited by a semicolon. By default, the variable **_fpath** contains the paths `lib/libc, lib/libch, lib/libopt,` and `libch/numeric` in the home directory of Ch. If the system variable **_fpath** is modified interactively in a Ch shell, it will be effective only for functions invoked in the current shell interactively. For running scripts, the setup of function search paths in the current shell will not be used and inherited in subshells. In this case, the system variable **_fpath** can be modified in startup file **_chrc** in Windows or **.chrc** in Unix in the user's home directory.

For example, if a file named `addition.chf` contains the program shown in Program 1, the function `addition()` will be treated as a system built-in function, which can be called to compute the sum $a + b$ of two input arguments a and b. Assume that the function file `addition.chf` is located at `C:/Documents and Settings/Administrator/c99/addition.chf`, the directory `C:/Documents and Settings/Administrator/c99` should be added to the function search path in the startup file **.chrc** in Unix or **_fpath** in Windows in the user's home directory with the following statement.

```
/* File: addition.chf
   A function file with file extension .chf */
int addition(int a, int b) {
    int c;
    c = a + b;
    return c;
}
```

Program 1: Function file `addition.chf`.

```
/* File: program.c
   Program uses function addition() in function file addition.chf */
#include <stdio.h>

/* This function prototype is optional when function addition() in
   file addition.chf is used in Ch */
int addition(int a, int b);

int main() {
    int a = 3, b = 4, sum;

    sum = addition(a, b);
    printf("sum = %d\n ", sum);
    return 0;
}
```

Program 2: A program using function file `addition.chf`.

```
    _fpath=stradd(_fpath, "C:/Documents and Settings/Administrator/c99;");
```

Function `addition()` then can be used either interactively in command mode as shown below,

```
> int i = 9
> i = addition(3, i)
12
```

or inside programs. In Program 2, the function `addition()` is called without a function prototype in the **main()** function so that the function prototype defined inside the function file `addition.chf` will be invoked. If the search paths for function files have not been properly setup, a warning message such as

```
WARNING: function 'addition()' not defined
```

will be displayed, when the function `addition()` is called.

When a function is called interactively in a Ch shell, the function file will be loaded. If you modify a function file after the function has been called, the subsequent calls in the command mode will still use the old version of the function definition that had been loaded. To invoke the modified version of the new function file, you can either remove the function definition in the system using the command **remvar** followed by a function name. or start a new Ch shell by typing **ch** at the prompt. For example, the command

```
> remvar addition
```

removes the definition for function `addition`(). The command **remvar** can also be used to remove a declared variable.

5.6 Interactive Execution of C++ Features

Not only C programs can be executed in Ch, but also classes and some C++ features are supported in Ch as shown below for interactive execution of C++ code.

```
> int i
> cin >> i
10
> cout << i
10
> class tagc {private: int m_i; public: void set(int); int get(int &);}
> void tagc::set(int i) {m_i = 2*i;}
> int tagc::get(int &i) {i++; return m_i;}
> tagc c
> c.set(20)
> c.get(i)
40
> i
11
> sizeof(tagc)
4
```

The input and output can be handled using **cin** and **cout** in C++. The public method `tagc::set()` sets the private member `m_i`, whereas the public method `tagc::get()` gets its value. The argument of method `tagc::get()` is passed by reference. The size of the class `tagc` is 4 bytes which does not include the memory for member functions.

6 Interactive Execution of Commands in the Input/Output Pane

Binary commands or C/C++ programs can also be executed interactively inside the input/output pane as shown in Figure 34. In Figure 34, the program `hello.c` is executed first in the input/output pane. Then, the command **pwd** prints the current working directory. The command **ls** lists files and directories in the current working directory. Options of a command can also be provided. For example, the command **ls** can invoked in the form of

```
ls -F
```

to list directories with a forward slash at the end.

To use a command with a complete path which containing a white space, the path needs to be placed inside a pair of double quotation marks, as shown below.

```
> "C:/Documents and Settings/Administrator/c99/hello.c"
```

How to execute /C/Ch/C++ programs with command line arguments is described in section 2.5.

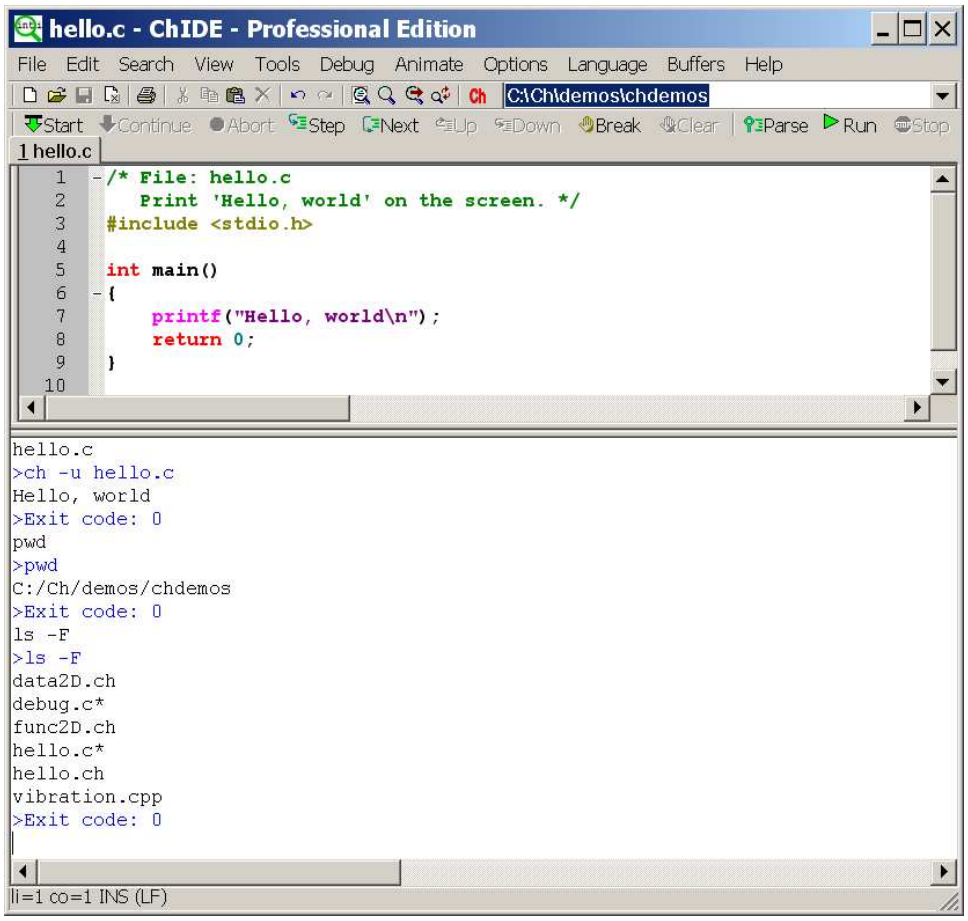


Figure 34: Executing commands inside the input/output pane.

7 Quick Animation

Ch can be used for quick animation using QuickAnimation^{TM}. It can also be used to display various objects based on specified x-y coordinate data. QuickAnimation^{TM} is especially suitable for animation of two-dimensional mechanical systems. Detailed information about QuickAnimation^{TM} can be found *QuickAnimation User's Guide* available in `CHHOME/docs/qanimate.pdf`.

A QuickAnimation file with the file extension `.qnm` can be edited in ChIDE with syntqax highlighting, as shown in Figure 35 for the QuickAnimation file `circles.qnm`. The animation for `circles.qnm` can be created by clicking the command `Animate->qanimate (Animate a qnm file)` or the function key F10, as shown in Figure 35. Figure 36 shows all frames for the animation.

A Ch program can be written to generate the standard output in the QuickAnimation format, using the standard C functions such as **printf()**. The standard output can be sent to the QuickAnimation program **qanimate** directly by clicking the command `Animate->Output to qanimate` or the function key F11, as shown in Figure 37 for running the program `CHHOME/demos/qanimate/smileyfaceanim.c`. Figure 38 shows a snapshot of the generated animation.

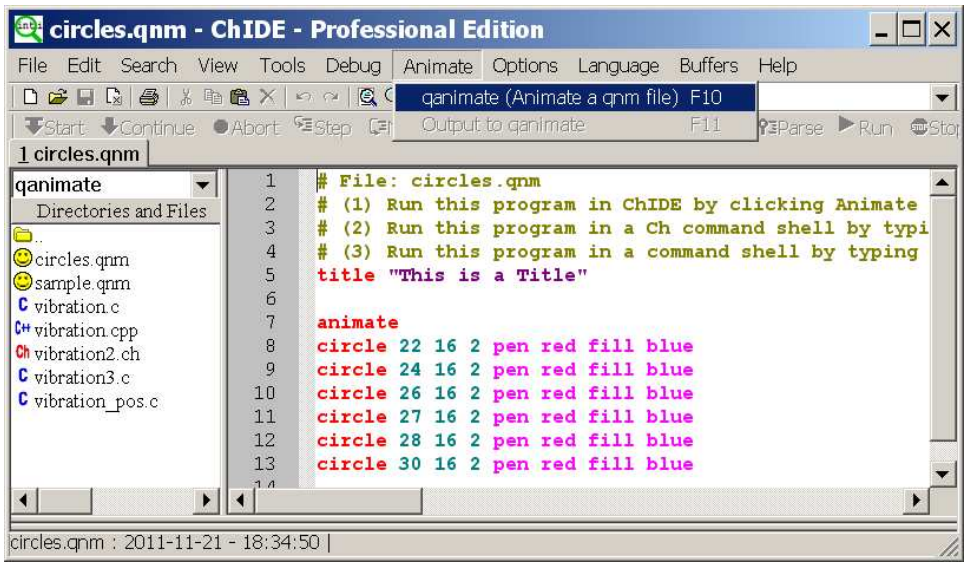


Figure 35: Executing a QuickAnimation file *circles.qnm*.

Figure 36: The output from executing the QuickAnimation file *circles.qnm* in Figure 35.

8 Compiling and Linking C/C++ Programs in ChIDE

ChIDE can also compile and link an edited C/C++ program in the editing pane using C and C++ compilers, then execute the created binary executable program. By default, the ChIDE is configured during the installation to use the latest Microsoft Visual Studio .NET installed in your Windows to compile C and C++ programs. The environment variables and commands for the Visual Studio compiler can be modified in the individual startup configuration file **_chrc** in the user's home directory, which can be opened for editing as shown in Figure 33. In Linux and Mac OS X x86, ChIDE uses compilers GNU gcc and g++ to compile C and C++ programs, respectively. The default compiler can be changed by modifying the C/Ch/C++ property file `cpp.properties` which can be opened by the command `Options->cpp.properties`.

The command `Tools->Compile` as shown in Figure 39 can be used to compile a program. The output and error messages for compiling a C or C++ program are displayed in the input/output pane of the ChIDE. In Windows, compiling a program will create an object file with file extension .obj. The object file can be linked using the command `Tools->Link` to create an executable program. The executable in Windows has file extension .exe.

If a make file makefile or Makefile is available in the current directory, the command `Tools->Build` will invoke the make file to build an application. A make file can also be invoked by right clicking the file name on the file tab, then clicking the command `make` in Linux or Mac and the command `make` or `nmake` in Windows as shown in Figure 40.

When ChIDE is used to edit a make file, the syntax will be highlighted. Because the tab character is reserved as a special character to begin a command for some make command, it will be preserved and not replaced with white spaces. A file with the file extension `.mak` or with the following file name is recognized

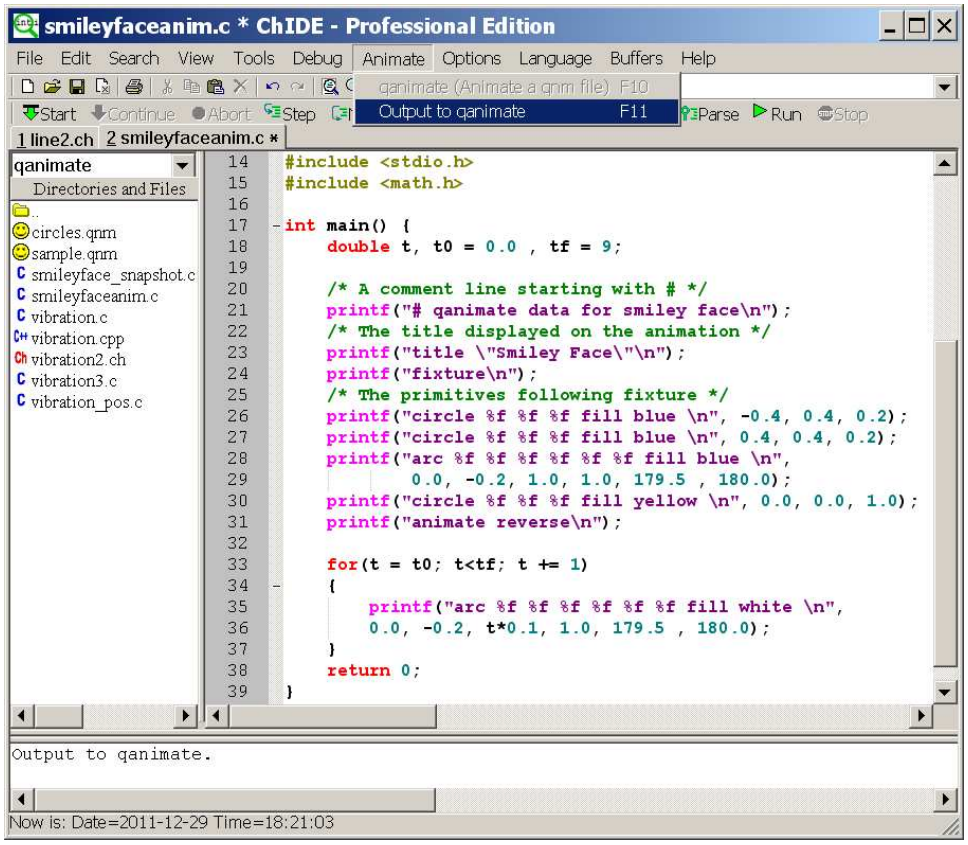


Figure 37: Executing a program with the standard output (stdout) sent to QuickAnimation.

Figure 38: The output from executing the program *smileyfaceanim.c* in Figure 37.

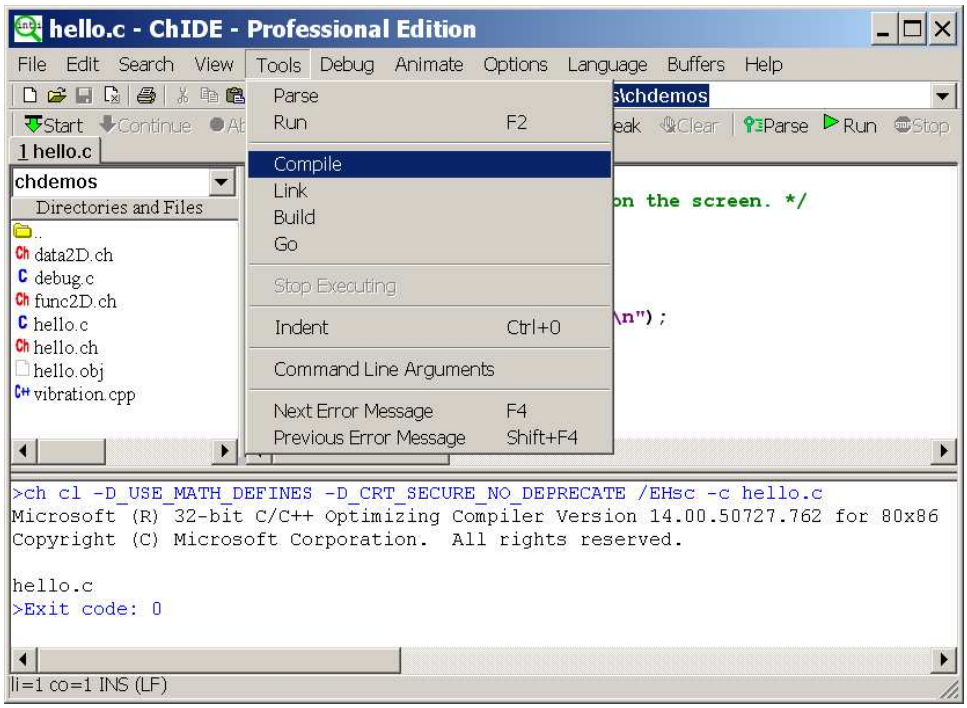


Figure 39: Compiling a C/C++ program.

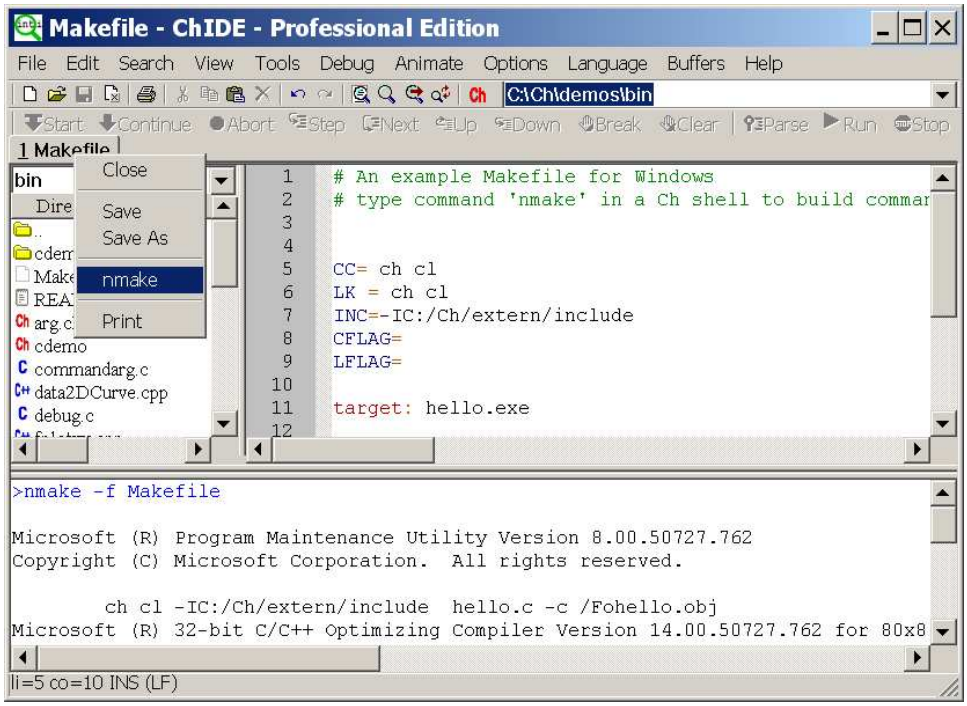


Figure 40: Using a makefile to compile a C/C++ program.

as a make file in ChIDE:

```
makefile
makefile.win
makefile_win
makefile.Win
makefile_Win
Makefile
Makefile.win
Makefile_win
Makefile.Win
Makefile_Win
```

The command `Tools->Go` will execute the developed executable program.

9 Other Computer Languages Understood by ChIDE

ChIDE is a general-purpose text editor. It currently is able to syntax highlighting the following languages.

- C/Ch/C++*
- CSS*
- HTML*
- Make
- SQL and PLSQL
- TeX and LaTeX
- XML*

If the symbol `'*'` is attached to a language, it denotes that the folding as described in section 3.5 is supported for the language.

Language settings are determined from the file extension but this can be changed by selecting another language from the `Language` menu.

10 Local Languages Supported in ChIDE

When Ch is installed in a platform in a language different from English, the menus and dialogs of ChIDE will be in its local language. By default, ChIDE supports more than 30 local languages as follows:

Afrikaans, Aribic, Basque, Brazilian Portuguese, Bulgarian, Catalan, Chinese Simplified, Chinese Traditional, Czech, Danish, Dutch, French, Galician, German, Greek, Hungarian, Indonesian, Italian, Japanese, Korean, Malaysian, Norwegian, Polish, Romanian, Portuguese, Russian, Serbian, Slovenian, Spanish, Spanish (Mexican), Swedish, Thai, Turkish, Ukrainian, and Welsh.

A new local language can also be easily supported.

Index

.chrc, 33
_chrc, 33
_fpath, 38
_ipath, 34
_path, 33, 34

abbreviations, 17
animation, 41

buffers, 21

cd, 31
ChIDE, 1
chide, 31
chmod, 35
chrc, 33
command shell, 30
commands, 40
compile, 42
Compile and Link Commands
    Build, 42
    Compile, 42
    Go, 42
    Link, 42
copyright, i
cp, 31
CSS, 45

Debug Command
    Watch, 27
Debug Commands
    Abort, 22
    Continue, 21
    Down, 22
    Next, 21, 22
    Parse, 5
    Run, 5
    Start, 21
    Step, 21, 22
    Stop, 5
    Up, 22
Debug Commands inside Debug Command Pane
    abort, 29
    assign, 24
    call, 24
    clear, 29
    clearfunc, 29
    clearline, 29
    clearvar, 29
    cont, 26
    down, 26
    expr, 24
    help, 24
    locals, 26
    next, 26
    print, 24
    remove, 26, 27
    remove expr, 26
    run, 25
    stack, 26
    start, 25
    step, 26
    stopat, 27
    stopin, 27
    stopvar, 27
    up, 26
    variables, 26
    watch, 26
Debug Console Window, 29
Debug Pane
    Breakpoints, 22
    Locals, 22
    Stack, 23
    Variables, 23
debugging, 21

edit, 14
Embedded Ch, 1

find, 15
folding, 16
font size, 16
function
    function files, 38
function keys, 16

homework, 17
HTML, 45
html, 45

IDE, 1
Input/Output Pane, 7
input/output pane, 40
Integrated Development Environment, 1

keyboard commands, 16

languages
    CSS, 45
    HTML, 45
    html, 45
    LaTeX, 45
    Make, 45
    PLSQL, 45

SQL, 45
Tex, 45
XML, 45
LaTeX, 45
link, 42
ls, 31

Make, 45
Makefile, 42
makefile, 42
mkdir, 31

Output, 7

PLSQL, 45
prompt, 30
pwd, 31

quick animation, 41
QuickAnimation, 41

remvar, 39
replace, 15
rm, 31

sessions, 21
SQL, 45
stradd(), 33, 34

Tex, 45

Unix Commands
cd, 31
cp, 31
ls, 31
mkdir, 31
pwd, 31
rm, 31
rmdir, 31

XML, 45