

# Getting Started with Ch IDE and Ch Command Shell

Ch Version 6.1

**func.c - ChIDE**

File Edit Search View Tools Debug Options Language Buffers Help

Start Continue Abort Step Next Up Down Break Clear Parse Run Stop

**1 func.c**

```

1  #include <stdio.h>
2
3  int i = 100;
4  int g = 200;
5  void func(int n) {
6      int i = 1;
7      double a[5] = {1,2,3,4,5};
8
9      g = 10;
10 }
11
12 int main() {
13     int i = 10;
14
15     func(i);
16     printf("Done!\n");
17     return 0;
18 }

```

Locals Variables Stack Watch Breakpoints

Name	Value
i	1
a	1.0000 2.0000 3.0000 4.0000 5.0000
n	10

debug> a  
a 1.0000 2.0000 3.0000 4.0000 5.0000  
debug> i  
i 1  
debug> 2\*g  
2\*g 400  
debug>

208 chars in 18 lines. Sel: 0 chars.

## 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](mailto:info@softintegration.com)

Copyright ©2001-2008 by SoftIntegration, Inc. All rights reserved.

Revision 6.1.0, September 2008

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, 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 95, Windows 98, Windows Me, Windows NT, Windows 2000, and Windows XP 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. All other trademarks belong to their respective holders.

## Table of Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Getting Started with Ch IDE</b>	<b>1</b>
<b>3</b>	<b>Debugging C/Ch/C++ Programs</b>	<b>6</b>
<b>4</b>	<b>Using Debug Commands Inside the Debug Command Window</b>	<b>12</b>
<b>5</b>	<b>Getting Started with Ch Command Shell</b>	<b>16</b>
5.1	Portable Commands for Handling Files. . . . .	17
5.2	Interactive Execution of Programs . . . . .	19
5.3	Setup Paths and Finding Commands in Ch . . . . .	19
5.4	Interactive Execution of Expressions and Statements . . . . .	21
5.5	Interactive Execution of Functions . . . . .	23
5.6	Interactive Execution of C++ Programming Features . . . . .	25
<b>6</b>	<b>Interactive Execution of Binary Commands in the Output Pane</b>	<b>25</b>
<b>7</b>	<b>Compiling and Linking C/C++ Programs</b>	<b>26</b>
<b>8</b>	<b>Commonly Used Keyboard Commands in ChIDE</b>	<b>26</b>

## 1 Introduction

Ch is an embeddable C/C++ interpreter. It is a superset of C with classes in C++ and other high-level extensions. Possible uses for Ch include but are not limited to cross-platform scripting, shell programming, 2D/3D plotting, numerical computing, and embedded scripting. Because Ch is interpretive, it allows C/C++ programs to be executed without compiling and linking. It is more suitable for interactive classroom presentations in teaching and for students learning C and C++. With advanced numerical features, it can be conveniently used for applications in engineering and science. This document presents a quick introduction on how to use this C/C++ interpreter using Ch IDE and Ch command shell.

## 2 Getting Started with Ch IDE

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 an Integrated Development Environment (IDE) to edit, debug, and run C/Ch/C++ programs in Ch interpretively without compilation. ChIDE can also compile and link edited C/C++ programs using C and C++ compilers of your choice such as Microsoft Visual Studio .NET. ChIDE is developed using Embedded Ch.

ChIDE is available in Windows for Ch Professional, Student, and Evaluation Editions.

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

Text editing in ChIDE works similarly to most Macintosh or Windows editors such as Notepad with the additional feature of automatic syntax styling. The user interface can be in one of 30 local languages such as German, French, Chinese, and Korean. 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 to be in memory at once.

As an example, open a new document, and type

```
#include <stdio.h>

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

in the text as shown in Figure 2 in the editing pane. The program appears colored due to syntax highlighting.

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.

Save the document as a file named `hello.c` as shown in Figure 3. The program `hello.c`, located in `CHHOME/demos/bin/hello.c` can also be loaded using `File | Open` command. Here `CHHOME` is the home directory for Ch. By default, the home directory for Ch in Windows is `C:/Ch` in the `C` drive.

Perform the `Run or Tools | Run` command as shown in Figure 4 to execute the program `hello.c`. Instead of performing the `Run or Tools | Run` command, pressing function key `F2` will also execute the program.



Figure 1. A ChIDE icon on a desktop in Windows.

## 2 GETTING STARTED WITH CH IDE

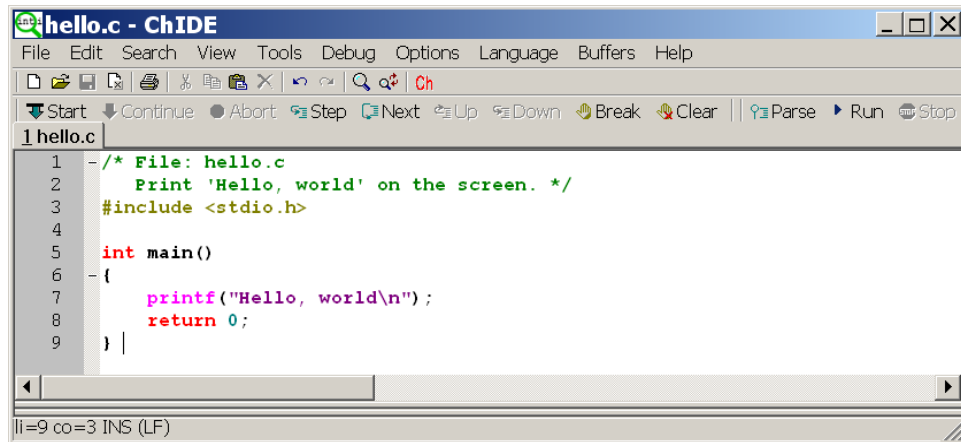


Figure 2. The program edited inside the editing pane in ChIDE.

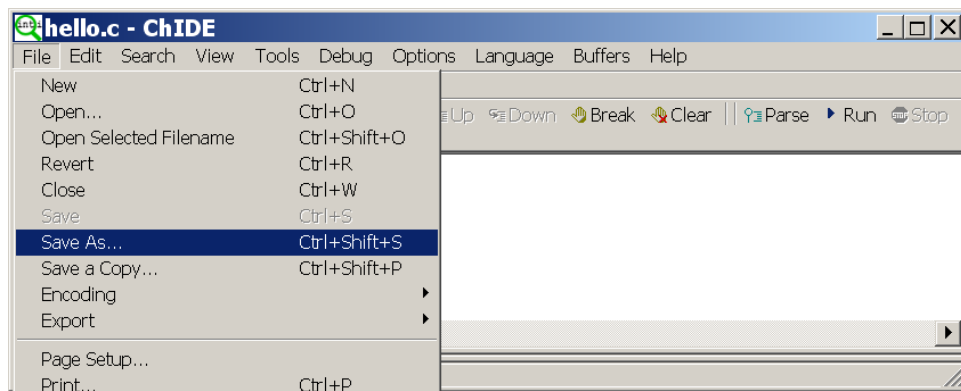


Figure 3. Save the edited program in ChIDE.

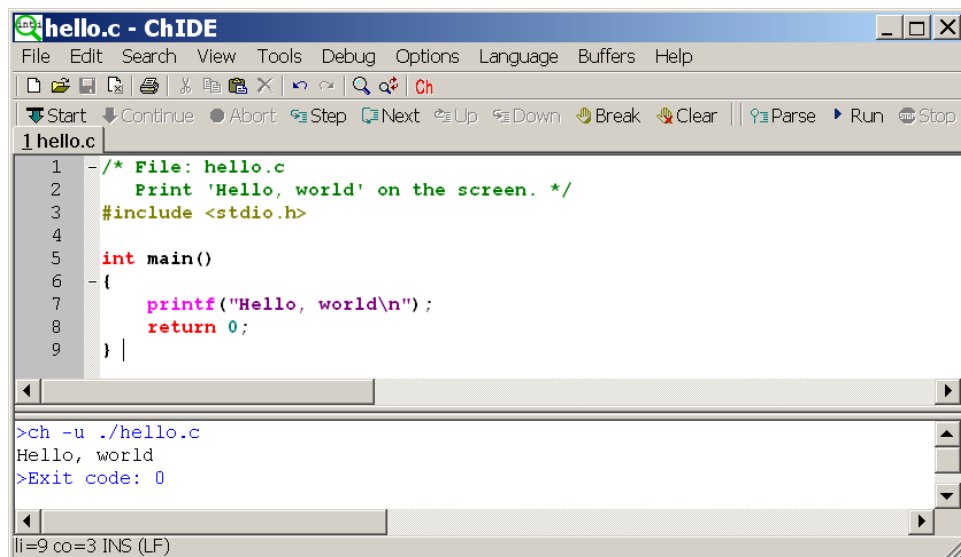


Figure 4. Run the program inside the editing pane in ChIDE and its output.

## 2 GETTING STARTED WITH CH IDE

There are four panes in ChIDE: the editing pane, debugging pane, debug command pane, and output pane. The debugging pane is located either to the below of the editing pane or on the right. 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 debugging pane or on the right. Similarly, the output pane is located either to the below of the debugging pane or on the right. The output pane is on the left of the debug command pane. Initially the output pane is of zero size, but it can also be made larger by dragging the divider between it and the debugging pane. By default, the output from the program is directed into the output pane.

The Options | Vertical Split command can be used to move the debugging pane to the right of the editing pane, followed by the output pane and debug command pane.

The same program `hello.c` in `CHHOME/demos/bin/hello.c`, where `CHHOME` is the home directory for Ch such as `C:/Ch` in Windows for `C:/Ch/demos/bin/hello.c`, can also be loaded using File | Open command.

When the program `hello.c` is executed, the output window 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 4. The first blue line

```
ch -u hello.c
```

from ChIDE shows that it uses Ch to execute the program `hello.c`. The black line is the output from running the Ch program. The last blue line 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 -10;
```

or

```
exit(-2);
```

the exit code would be -1.

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";
```

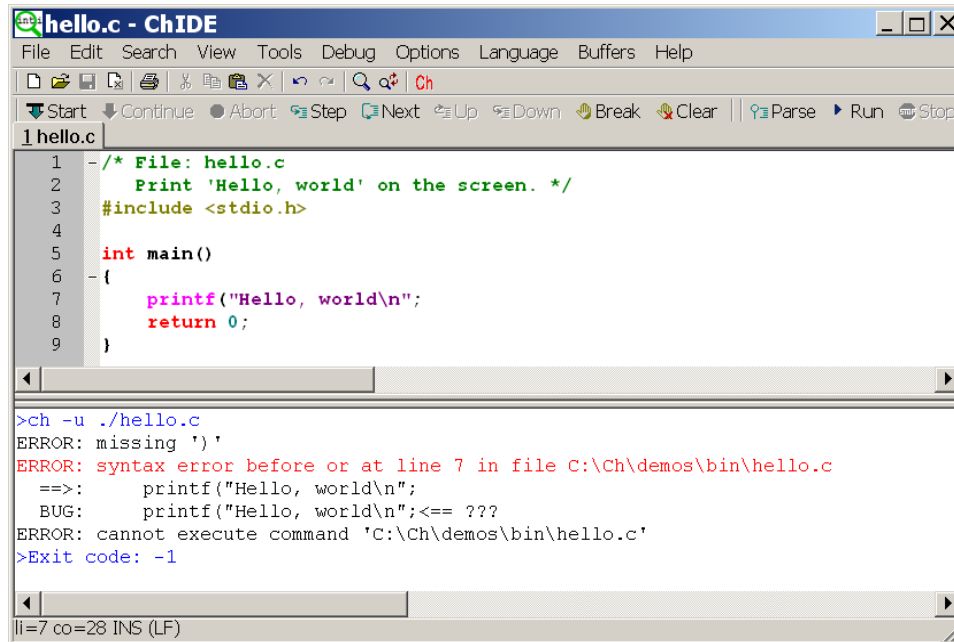


Figure 5. The error line in output from executing program hello.c.

Perform the **Run** or **Tools | Run** command for the modified program. The results should look similar to those below

```
ERROR: missing ')'
```

```
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'
```

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

as shown in Figure 5. Because the program fails to execute, the exit code -1 is displayed at the end of the output pane as

```
Exit code: -1
```

If you double click the red colored error message in the output pane shown in Figure 5 with the left button of your mouse, the line with incorrect syntax and the error message in the output pane will be highlighted with a yellow background as shown in Figure 6. The caret is moved to this line and the pane is automatically scrolled if needed to show the line. While it is easy to see where the problem is in this simple case, with a large file, the **Tools | Next Message** command can be used to view each of the reported errors. Upon performing **Tools | Next Message**, the first error message in the output pane and the appropriate line in the editing pane are highlighted with a yellow background.

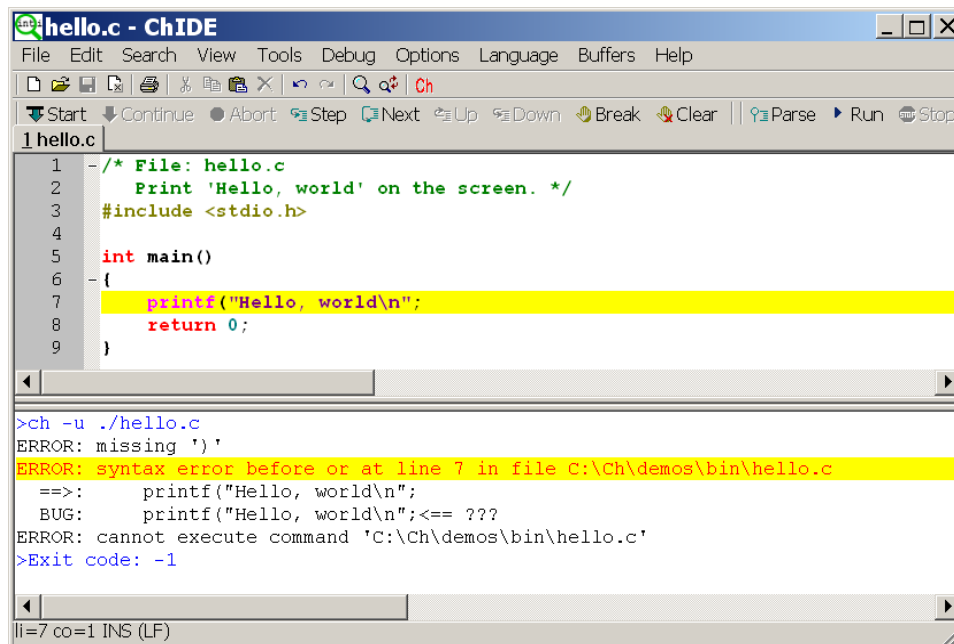
The output window can be opened and closed by the command **View | Output Window**. The contents of the console window can be cleared by the command **View | Clear Output Window** as shown in Figure 7.

If command execution has failed and is taking too long to complete, then the **Stop** or **Tools | Stop Executing** command, or function key F4, can be used to stop the program.

You may use command **Parse** or **Tools | Parse** to just check the syntax error of the program without executing it.

ChIDE can also execute programs that require the user's input through such C functions as `scanf()`. It can also handle command parameters. More information about running C and C++ programs in Ch using ChIDE can be obtained on-line by clicking ChIDE Help from the Help menu as shown in Figure 8.

## 2 GETTING STARTED WITH CH IDE



The screenshot shows the ChIDE interface with the file `hello.c` open. The code is as follows:

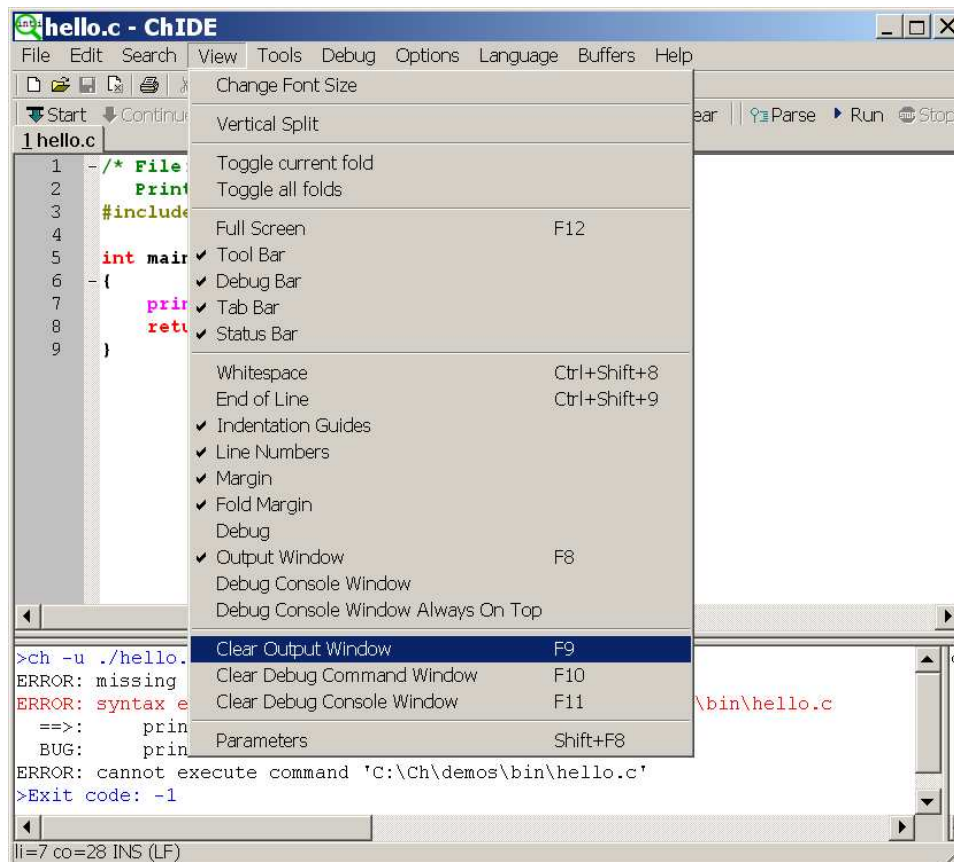
```
1  /* File: hello.c
2     Print 'Hello, world' on the screen. */
3     #include <stdio.h>
4
5     int main()
6     {
7         printf("Hello, world\n");
8         return 0;
9     }
```

The output window at the bottom shows the following error message:

```
>ch -u ./hello.c
ERROR: missing ')'
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'
>Exit code: -1
```

The error message is highlighted in yellow. The status bar at the bottom indicates `li=7 co=1 INS (LF)`.

Figure 6. Finding the error line in output from executing program `hello.c`.



The screenshot shows the ChIDE interface with the `View` menu open. The menu options are as follows:

- Change Font Size
- Vertical Split
- Toggle current fold
- Toggle all folds
- Full Screen (F12)
- Tool Bar (checked)
- Debug Bar (checked)
- Tab Bar (checked)
- Status Bar (checked)
- Whitespace (Ctrl+Shift+8)
- End of Line (Ctrl+Shift+9)
- Indentation Guides (checked)
- Line Numbers (checked)
- Margin (checked)
- Fold Margin (checked)
- Debug (checked)
- Output Window (checked) (F8)
- Debug Console Window
- Debug Console Window Always On Top
- Clear Output Window (F9)
- Clear Debug Command Window (F10)
- Clear Debug Console Window (F11)
- Parameters (Shift+F8)

The `Clear Output Window` option is highlighted. The output window at the bottom shows the same error message as in Figure 6.

Figure 7. Clearing the contents in the output window.



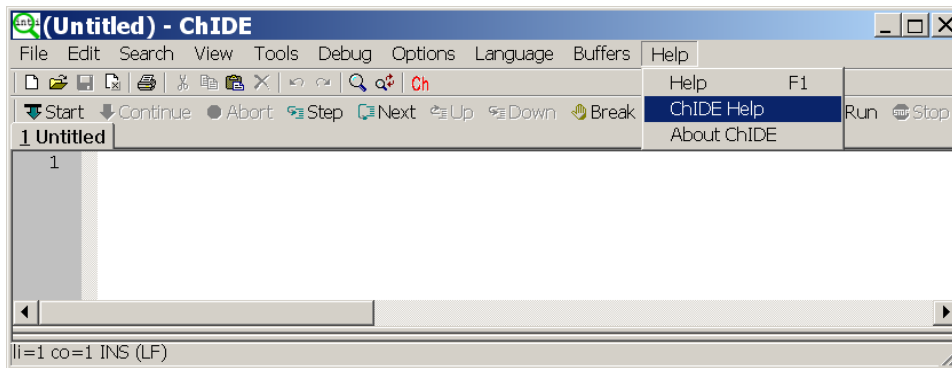


Figure 8. Get on-line help on how to use ChIDE.

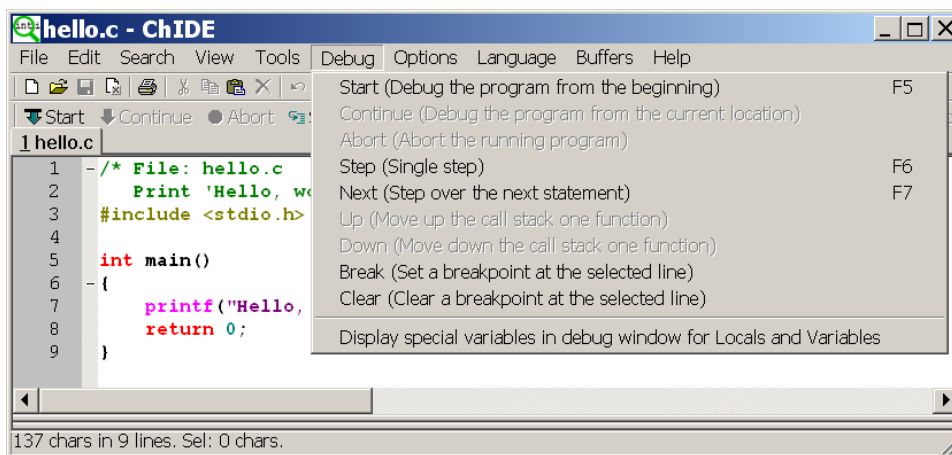


Figure 9. Debug menus.

### 3 Debugging C/Ch/C++ Programs

The Ch IDE has all capabilities available in a typical debugger for binary C programs. The debug interface commands, such as *Start* and *Clear*, are shown in Figure 9.

They are also available directly on the debug bar as shown in Figure 10. The applicable commands in the debug bar at any point of debugging will be clickable. Non-clickable commands are dimmed.

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 it hits a breakpoint or the program ends.

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 10. 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* above the debug window as shown in Figure 10. The debug window will display the breakpoint number and its location for each breakpoint. A breakpoint for the current line can also be added by clicking the menu *Break*. It can also be deleted by clicking the menu *Clear*. 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;
```

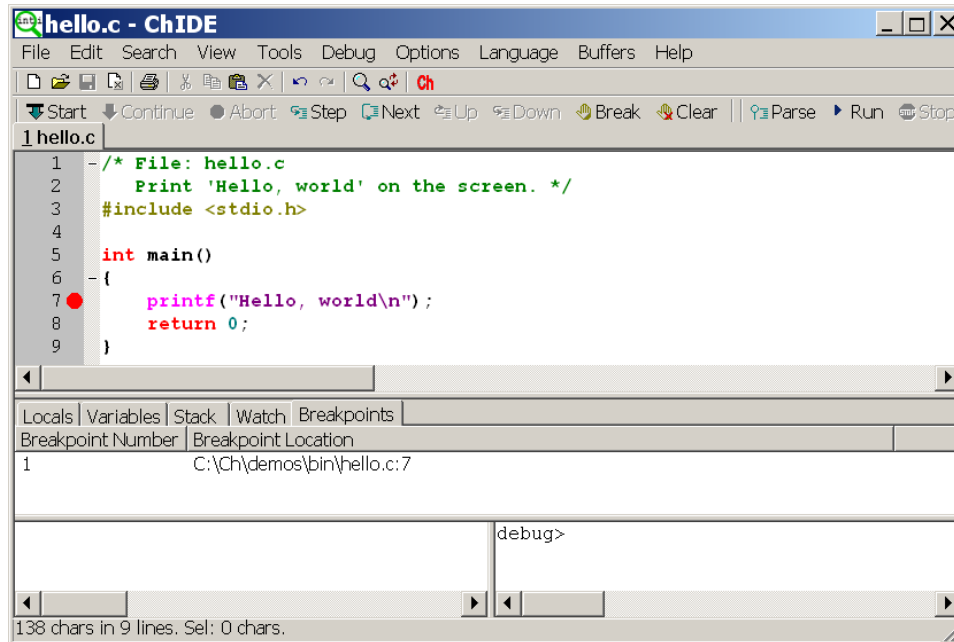


Figure 10. Set a breakpoint.

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. However, when 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 window, a breakpoint can also be set for functions and controlling variables, which will be described later.

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

When a program is executed in the debug mode, the standard input, output, and error streams are redirected in a separate Debug Console Window shown in Figure 11. 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 console window can be opened and closed by the command **View | Debug Console Window**. The contents of the console window can be cleared by the command **Debug | Clear Debug Console Window** as shown in Figure 7. The colors for background and text as well as the windows size and font size of the console window can be changed by right clicking the ChIDE icon on the upper left corner of the window and selecting **Properties** menu to make changes. Note that for Windows Vista, you need to run ChIDE with the administrative privilege to make such a change.

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 window by clicking menu **Locals** above the debug window. When control of the program execution is inside a function, 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 12, 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.

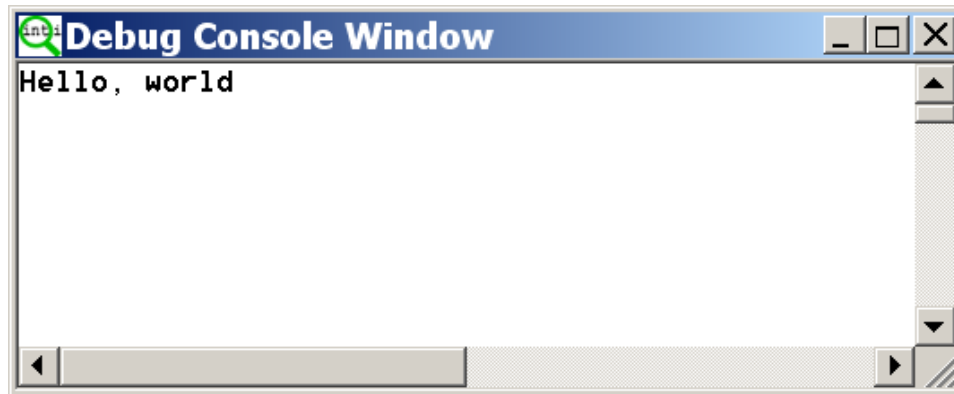


Figure 11. Debug Console Window for Input/Output in Debugging.

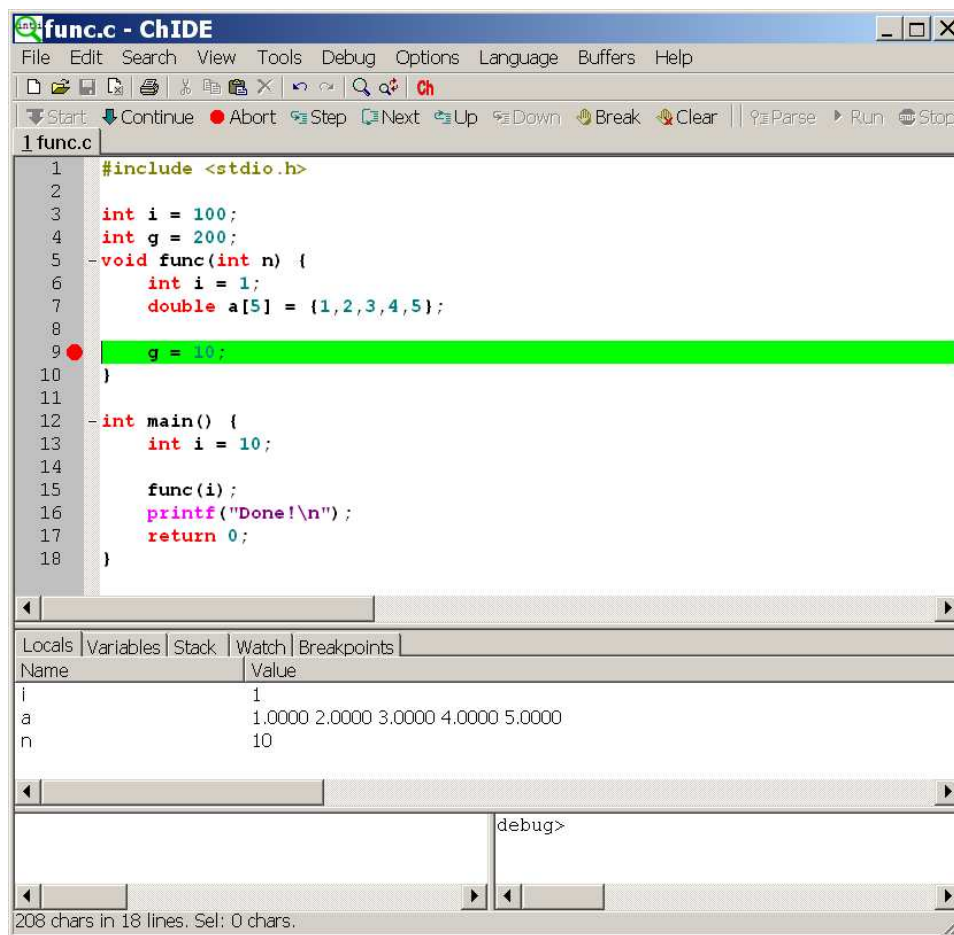


Figure 12. Display names and values of local variables in the currently called function.

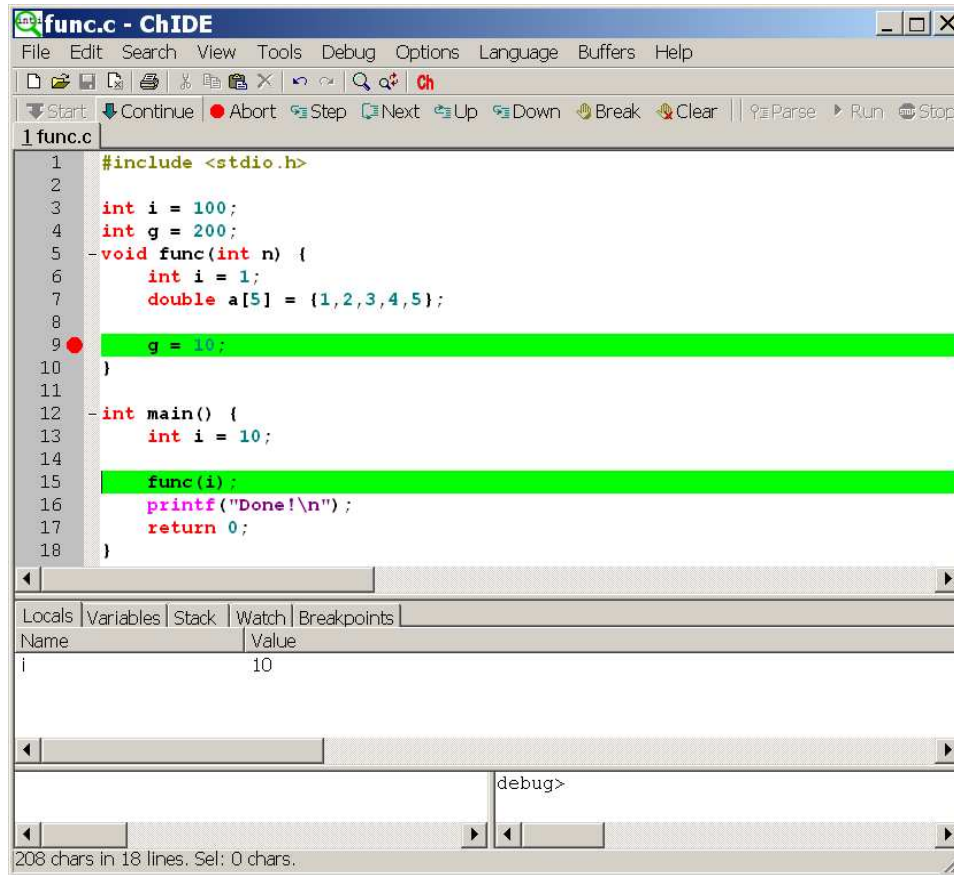


Figure 13. Display names and values of local variables in the calling function.

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 or accessed in the debug window. For example, when clicking command Up in Figure 12, the control flow of the program moves to its calling function `main()` at line 15 as highlighted with the color green also in Figure 13. The menu Down as shown in Figure 12 is not clickable. But, the menu Down is clickable in Figure 13 when the current stack is moved up. The debug window at this point displays the name and value of the variable `i`, the only regular variable, in the calling function `main()`.

Command Stack above the debug command 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 14, function `func()` is called by function `main()`, which in turn is invoked by program `func.c` located in the directory `C:\Ch\demos\bin\func.c`.

Names and their corresponding values of variables in all stacks can be displayed by the command Variables above the debug window as shown in Figure 15. In this case, 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 window. 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 window for Locals and Variables in the debug menu shown in Figure 9 is clicked, names and values of special variables such as `__func__` will be displayed in the debug window for commands Locals and Variables.

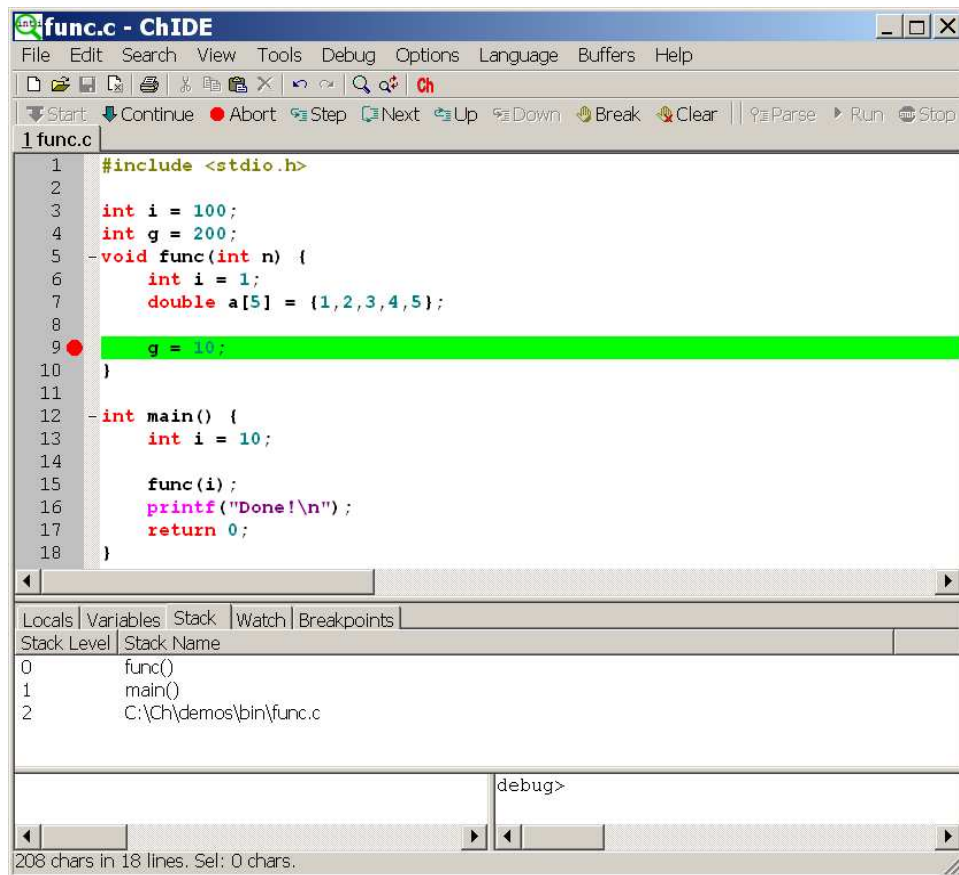


Figure 14. Display different stacks for the executing point.

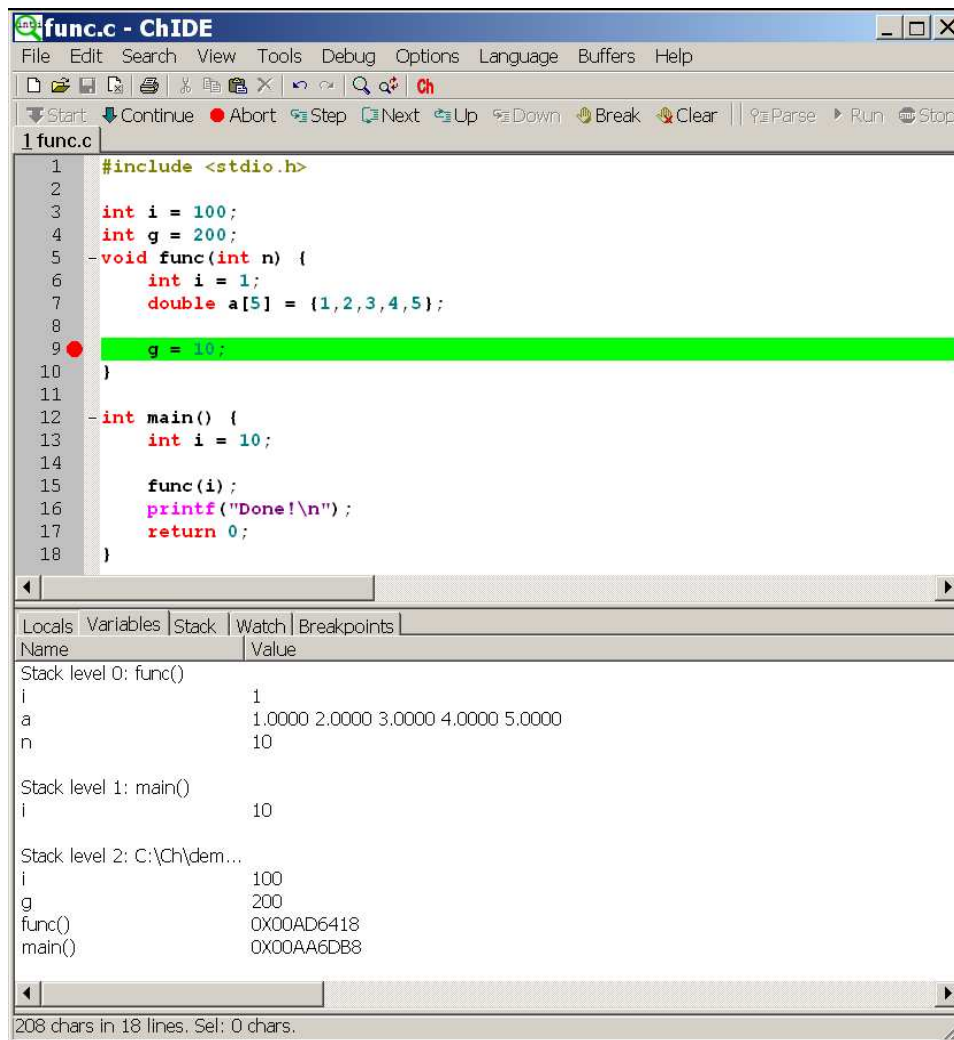


Figure 15. Display names and values of all variables in all stacks .



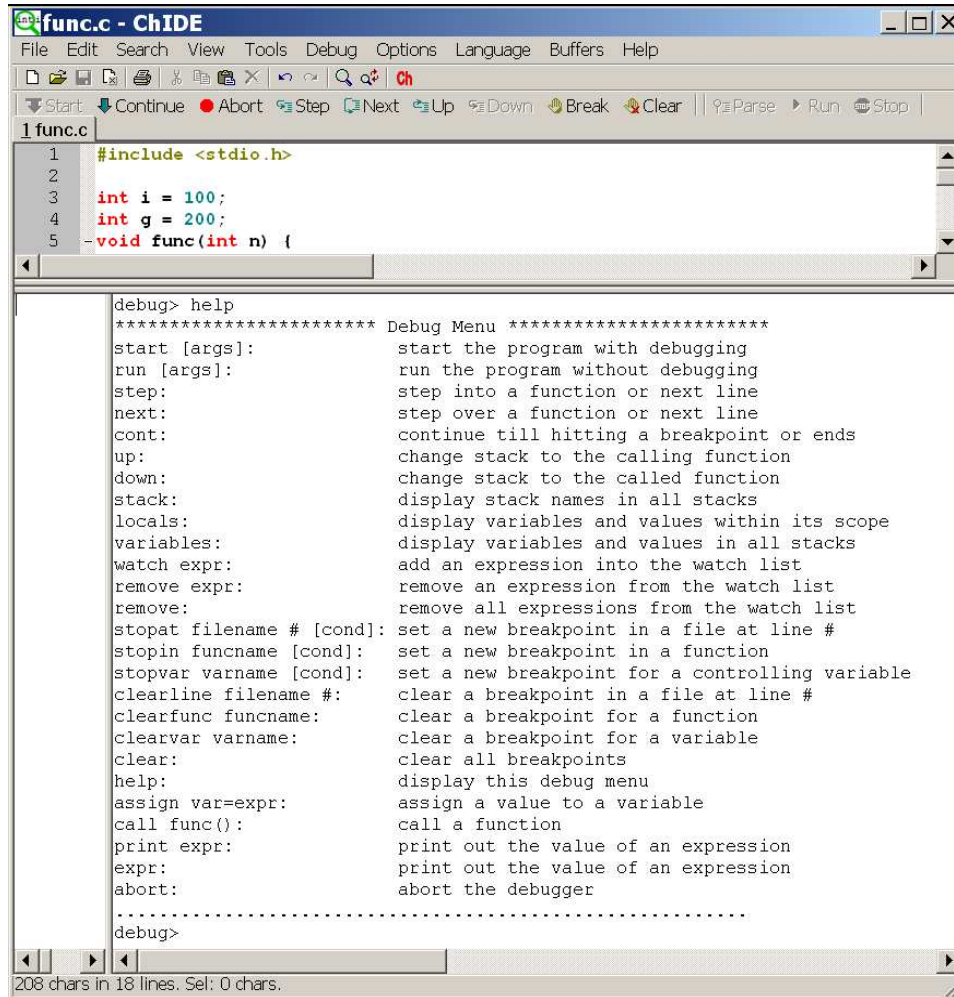


Figure 16. Debug commands in the debug command window.

## 4 Using Debug Commands Inside the Debug Command Window

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

```
debug>
```

inside the debug command window indicates that the debugger is ready to accept debug commands. Type the command `help`, it will display all available commands as shown in Figure 16. 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 in the debug bar are available in this interactive debug command window. However, some features are available only through the debug command window.

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
```

#### 4 USING DEBUG COMMANDS INSIDE THE DEBUG COMMAND WINDOW

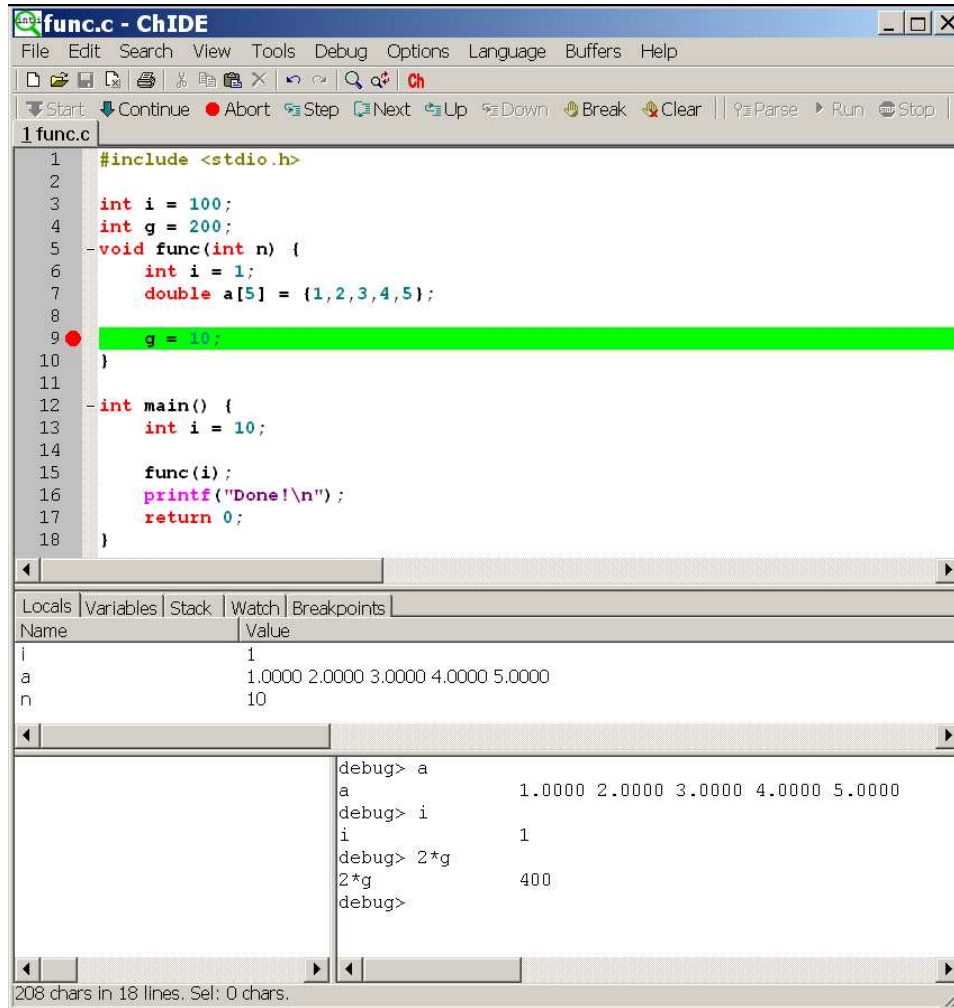


Figure 17. Using debug commands in the debug command window.

```
debug> call func()
debug> print i
20
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 17, the values of variables `a` and `i` as well as expression `2*g` can be obtained by typing corresponding commands in the debug command window.

Command `start` begins debugging a program. The optional arguments for the command `start` and `run` are processed and passed to the arguments for function `main()`. For example, to run program `C:\Ch\demos\bin\commandarg.c` shown in Figure 18, the debug command

```
debug> start -o option1 -v option2 option3 option4
```

will assign the strings `"C:\\Ch\\demos\\bin\\commandarg.c"`, `"-0"`, `"option1"`, `"-v"`, `"option2"`, `"option3"`, and `"option4"` to elements `argv[0]`, `argv[1]`, `argv[2]`, `argv[3]`, `argv[4]`, `argv[5]`, and `argv[6]` of the argument `argv` of the `main` function



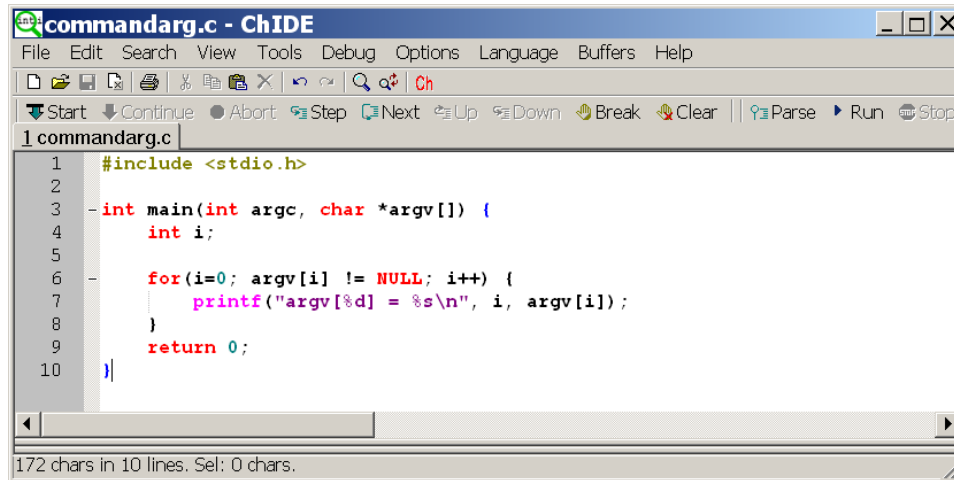


Figure 18. A program for handling command parameters.

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

of the Ch script `commandarg.c`, respectively. An optional argument with space should be enclosed within two double quotation marks as shown below.

```
debug> start opt1 "opt2 with space" opt3
```

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 or 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 window. 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 window

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

add expression `2*g` and variable `i` to a list of watched expressions as shown in Figure 19. 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 window by clicking the command `Watch` above the debug window as shown in Figure 19.

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.

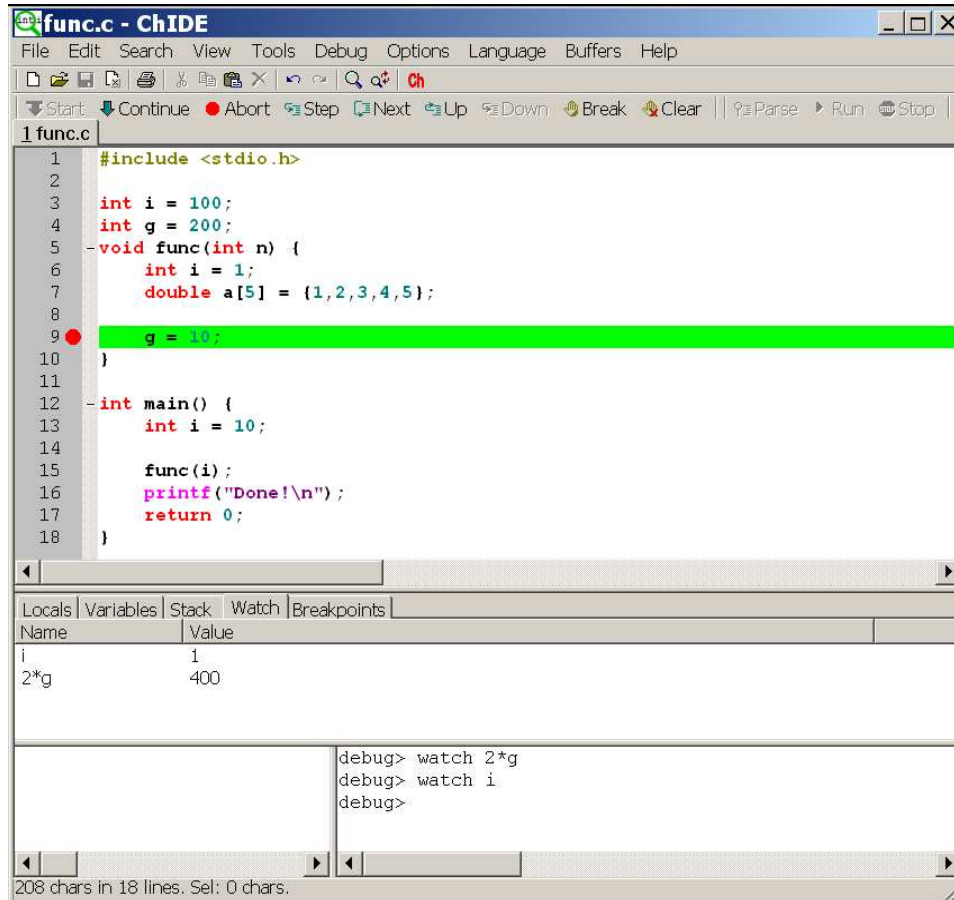


Figure 19. Set watch expressions and variables inside the debug command window to display their values in the debug window.

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, 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

```

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

```



Figure 20. A Ch icon on a desktop in Windows.

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
```

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.ch` 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 the 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 window can be cleared by clicking the command View | Clear Debug Command Window as shown in Figure 7.

## 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. Therefore, the Ch command shell 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.

A Ch shell can be launched by running the command `ch`. In Windows, a Ch command shell can also be conveniently launched by clicking the red-colored **Ch** icon, shown in Figure 20, on the desktop or on the toolbar 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 21. The colors of the text and background as well as the window size and font

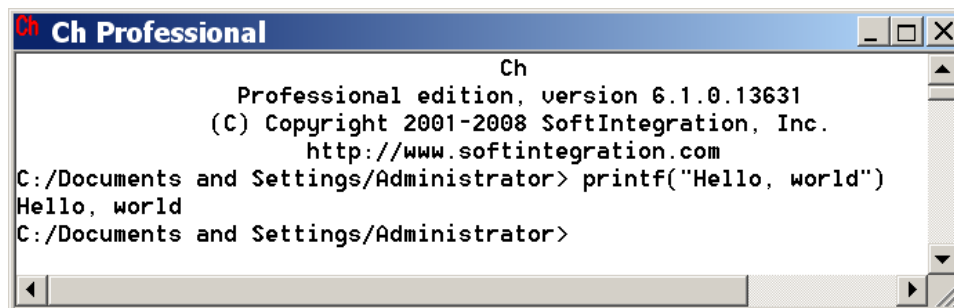


Figure 21. A Ch command shell.

size of the shell window can be changed by right clicking the Ch icon at the upper left corner of the window, and select Properties menu 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 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 21.

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

In comparison with Figure 21, 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 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 1, 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 or Linux. 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
```

Table 1. Portable commands for handling files.

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

```

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 1, 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 2 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 Ch IDE as shown in Figure 2.

## 5.2 Interactive Execution of 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 the previous section. 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, 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.

## 5.3 Setup Paths and Finding Commands 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 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, functions, header 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



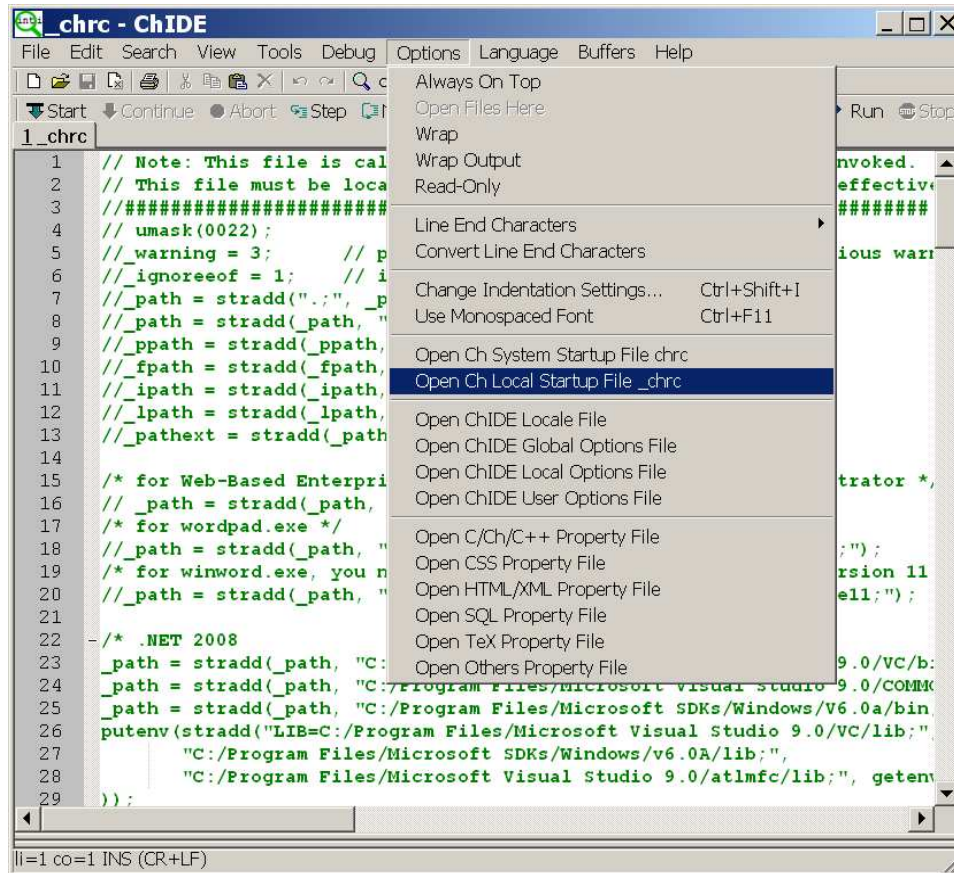


Figure 22. Open the local Ch initialization startup file for editing.

```
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 for editing the search paths by ChIDE editor as shown in Figure 22.

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

```
_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
```

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 for function files by the statement

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

A function file contains the function definition.

In Unix, 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`.

## 5.4 Interactive Execution of 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:

```
> 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:

```
> 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. Different format specifiers for the families of input function `fscanf()` and output function `fprintf()` using file streams opened by function `fopen()` can also be tried out this way.

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
10.00000
```

All C operators can be used interactively as shown:

```
> 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:

```
> 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`. 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:

```
> 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 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; return a+b+sin(1.5);}
> double c
> c = add(10.0, 20)
30.9975
```

```

/* 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.

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 at 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.

```
_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. The output of Program 2 is `c = 5`. 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.

```

/* 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. Program using function file addition.chf.

## 5.6 Interactive Execution of C++ Programming 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 Binary Commands in the Output Pane

Binary commands can also be executed interactively inside the output pane as shown in Figure 23. In Figure 23, command `pwd` in the output pane prints the current working directory. 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 be invoked in the form of

```
ls -F
```

to list directories with a forward slash at the end.

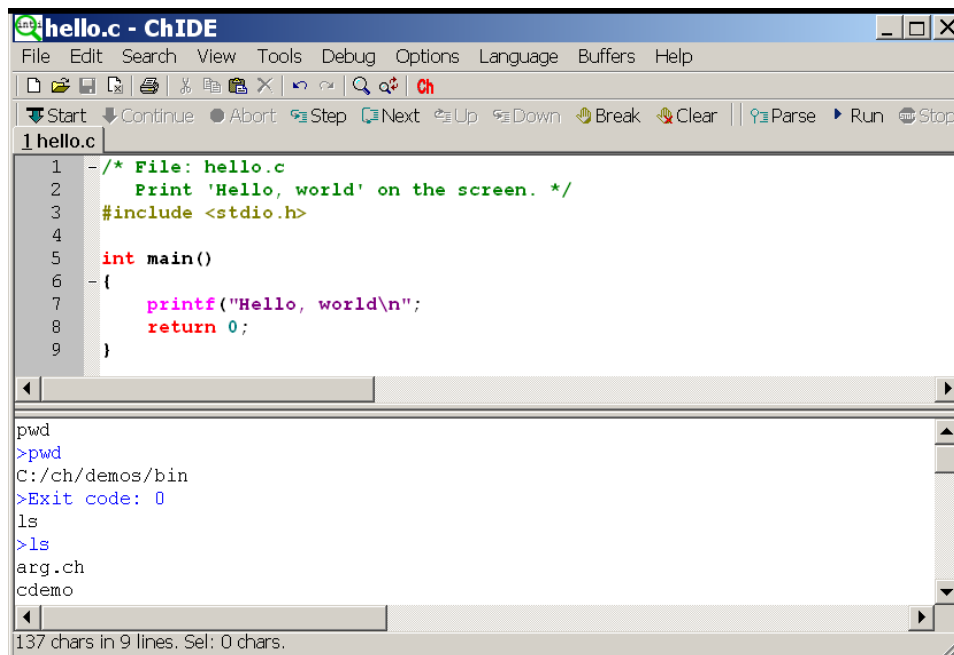


Figure 23. Execute commands inside the output pane.

## 7 Compiling and Linking C/C++ Programs

ChIDE can also compile and link an edited C/C++ program in the editing pane using C and C++ compilers. By default, the ChIDE is configured 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 22. In Linux, ChIDE uses compilers `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 under the command Options.

The command `Tools | Compile` as shown in Figure 24 can be used to compile a program.

The output and error messages for compiling a C or C++ program are displayed in the output window 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 Makefile is available in the current directory, the command `Tools | Build` will invoke the Makefile to build an application. The command `Tools | Go` will execute the developed executable program.

## 8 Commonly Used Keyboard Commands in ChIDE

Keyboard commands in ChIDE mostly follow common Windows and GTK+ conventions. All move keys (arrows, page up/down, home and end) allows to extend or reduce the stream selection when holding the Shift key, and the rectangular selection when holding the Shift and Alt keys. Keyboard equivalents of menu commands are listed in the menus. Figure 2 shows the most commonly used commands and their corresponding keyboard commands.

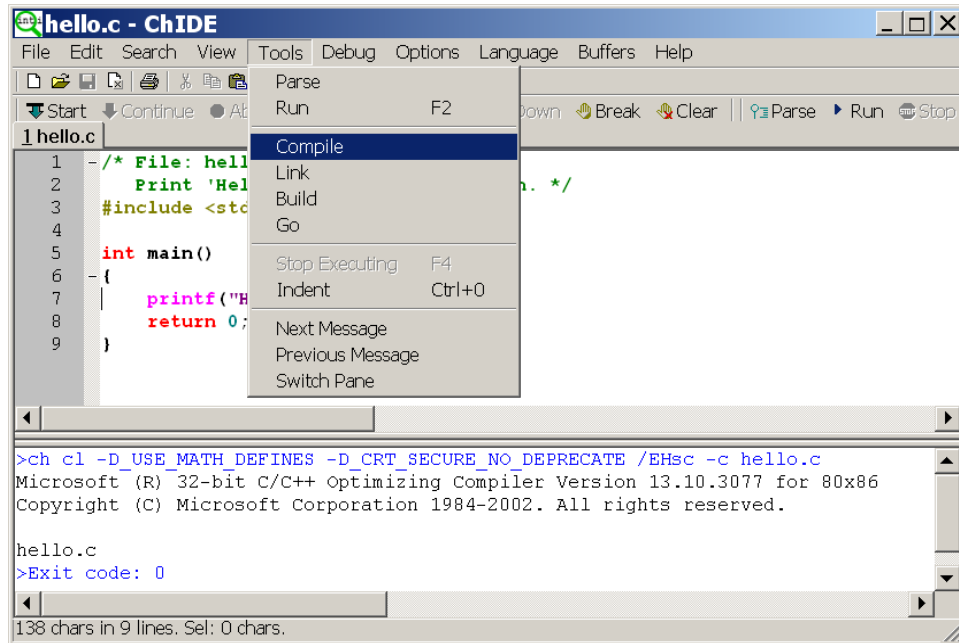


Figure 24. Compile a C/C++ program.

Table 2. 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
Stop Executing C/Ch/C++ program	F4
Start (Debug the program)	F5
Step (Single step)	F6
Next (Step over the next statement)	F7
Close/Open Output Window	F8
Clear Output Window	F9
Clear Debug Command Window	F10
Close/Open Debug Console Window	F11
Full screen	F12

## Index

- .chrc, 19
- \_chrc, 19
- \_fpath, 24
- \_ipath, 21
- \_path, 20, 21
  
- cd, 17
- ChIDE, 1
- chide, 17
- chmod, 19
- chrc, 19
- commands, 25
- compile, 26
- Compile and Link Commands
  - Build, 26
  - Compile, 26
  - Go, 26
  - Link, 26
- copyright, i
- cp, 17
  
- Debug Commands
  - Abort, 7
  - Continue, 6
  - Down, 7
  - Next, 6, 7
  - Parse, 4
  - Run, 1
  - Start, 6
  - Step, 6, 7
  - Stop, 4
  - Up, 7
- Debug Commands inside Debug Command Window
  - abort, 16
  - assign, 12
  - call, 12
  - clear, 16
  - clearfunc, 16
  - clearline, 16
  - clearvar, 16
  - cont, 14
  - down, 14
  - expr, 12
  - help, 12
  - locals, 14
  - next, 14
  - print, 12
  - remove, 14
  - remove expr, 14
  - run, 14
  - stack, 14
  - start, 13
  - step, 14
  - stopat, 15
  - stopin, 15
  - stopvar, 15
  - up, 14
  - variables, 14
  - watch, 14
- Debug Window
  - Breakpoints, 6
  - Locals, 7
  - Stack, 9
  - Variables, 9
  - Watch, 14
  
- Embedded Ch, 1
  
- function
  - function files, 24
- function keys, 26
  
- IDE, 1
- Integrated Development Environment, 1
  
- keyboard commands, 26
  
- link, 26
- ls, 17
  
- mkdir, 17
  
- Output, 4
- Output Window, 4
  
- prompt, 16
- pwd, 17
  
- remvar, 24
- rm, 17
  
- stradd(), 20, 21
  
- Unix Commands
  - cd, 17
  - cp, 17
  - ls, 17
  - mkdir, 17
  - pwd, 17
  - rm, 17
  - rmdir, 17