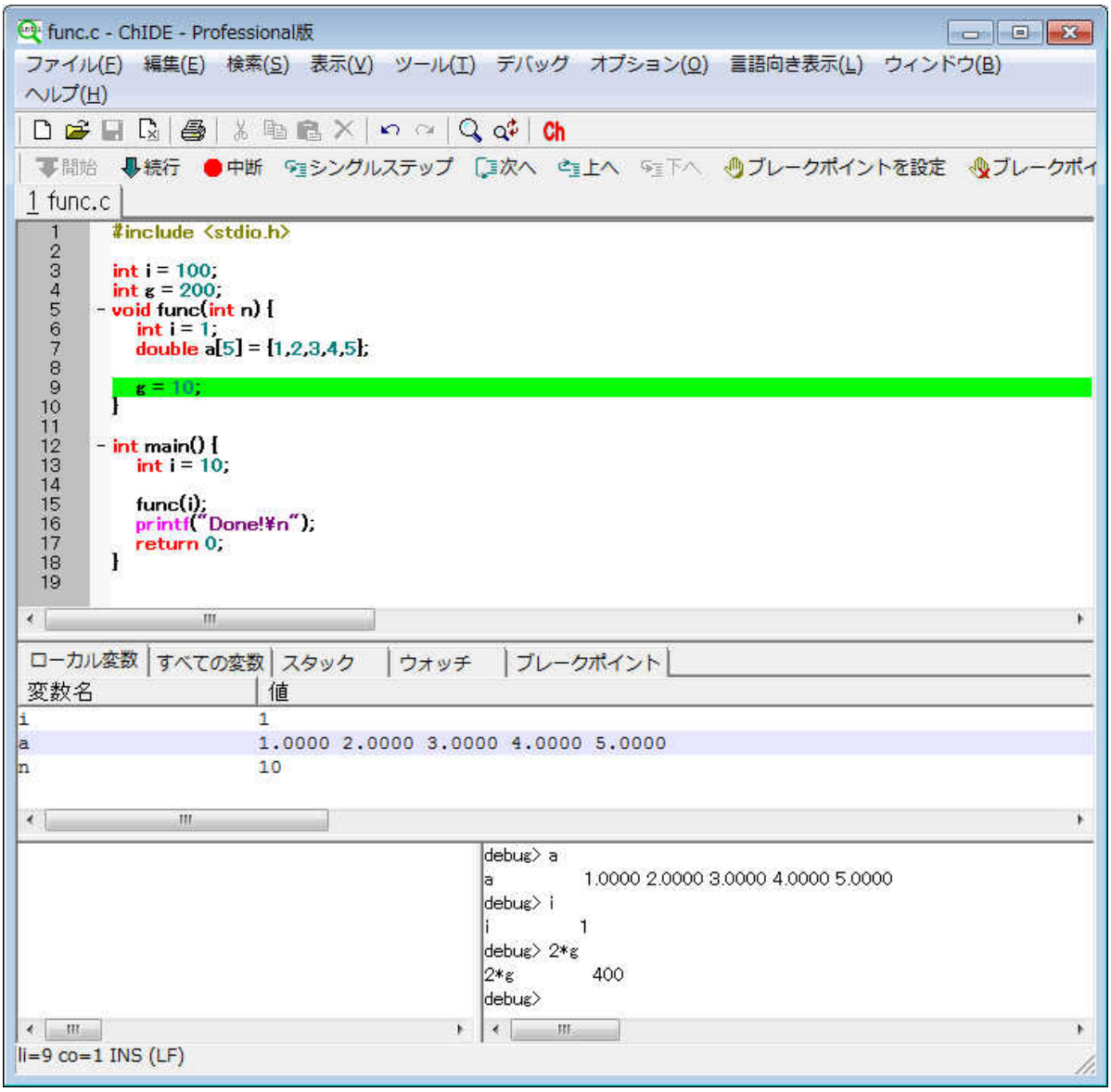
ChIDE および
Ch コマンドシェル入門

Ch バージョン6.3



SoftIntegration 社の連絡先

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

2010 年 2 月 日本語版 6.3

目次

1　はじめに

Ch は、組み込み可能な C/C++インタープリタです。Ch は、C++におけるクラスと他の高級言語拡張をもったスーパーセットです。Ch のユーザー層は、クロスプラットフォームスクリプティング、シェルプログラミング、2D/3D プロット、数値計算、および組み込みスクリプティングに限定されません。Ch はインタープリタですので、作成した C/C++プログラムを、コンパイル、リンクすることなしに実行することができます。Ch は、学級における学生への C および C++授業のプレゼンテーションに最適です。高度な数値演算機能をもつことにより、Ch は工学および自然科学のアプリケーションでの使用に適しています。この ドキュメント は、ChIDE と Ch コマンドシェルを使用した、C/C++インタープリタの使い方の手っ取り早い習得方法について述べています。

2　ChIDE 入門

統合化開発環境（IDE）は、C および C++プログラムの使用することができます。これは、特に自動書式ハイライト機能をもったプログラムの編集と IDE 内でのプログラムの実行に使用されます。ChIDE は、C/Ch/C++プログラムをコンパイルすることなしに、Ch で作成、デバッグ、実行するための統合化開発環境（IDE）です。ChIDE では、Microsoft Visual Studio .NET のような、C および C++コンパイラをオプションで用いて、編集済みの C/Ch/C++プログラムをコンパイル、リンクすることもできます。

ChIDE は、Embedded Ch を用いて開発されています。

Ch Professional 用の ChIDE は、Windows、Linux、および Mac OS X x86 [1]で使用可能です。

ChIDE は、コマンド `chide` を実行して起動することができます。Windows 上では、ChIDE は、デスクトップ上で図 1 に示されているアイコンをダブルクリックして、簡単に起動することも可能です。

Mac OS X x86 上では、ChIDE は、ダッシュボード上または Application フォルダ内の図 1 に示されているアイコンをクリックして起動することも可能です。

図 1: Windows、Linux、Mac OS X x86 デスクトップ上の ChIDE アイコン

Linux 上では、ChIDE はスタートアップメニュー内のエントリ Programming Tools から起動することも可能です。コマンド

```
ch -d
```

を使用すると、Ch のアイコンがデスクトップ上に作成されます。Ch が ChIDE とともにインストールされると、ChIDE のアイコンもまたデスクトップ上に作成されます。

ChIDE におけるテキスト編集作業は、[メモ帳] に自動書式スタイリング機能を追加したような、ほとんどの Windows または Macintosh エディタでの作業と類似しています。ChIDE は一度に複数のファイルをメモリ上に保持することが可能ですが、表示されるのは 1 つのファイルのみです。ChIDE は最大 20 個のファイルまでをメモリに保持可能です。

[1]現在のところ、日本語版として提供されているのは Windows 版のみです。

2　*CHIDE* 入門

1 つの例として、新規のドキュメントを開き、編集ウィンドウ内に図 2 のように、以下のコードを入力します。

```
#include <stdio.h>

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

プログラムは、書式ハイライト機能により、色付きで表示されます。

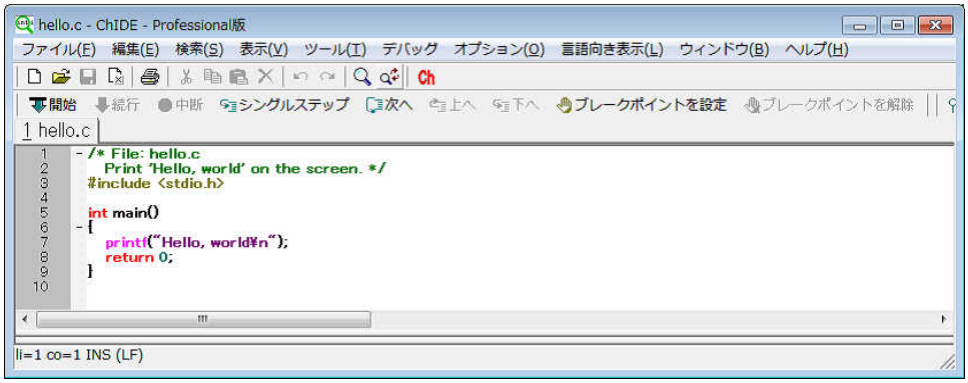


図 2: ChIDE の編集ウィンドウ内で編集されたプログラム

学級でのプレゼンテーション用に、表示されるプログラムのフォントサイズを大きくすることができます。変更を行うには、コマンド［表示］ → ［フォントサイズの変更］をクリックします。編集ウィンドウの左側の行番号、余白、および折りたたみ表示用余白は、コマンド［表示］ → ［行番号］、［余白］、［折りたたみ表示用の余白］をそれぞれクリックすることにより、図 3 のように、オフにすることができます。折りたたみ表示用余白上の折りたたみ点のマーカー '-' と '+' をクリックして、ブロックコードをそれぞれ展開したり、折りたたんだりすることができます。

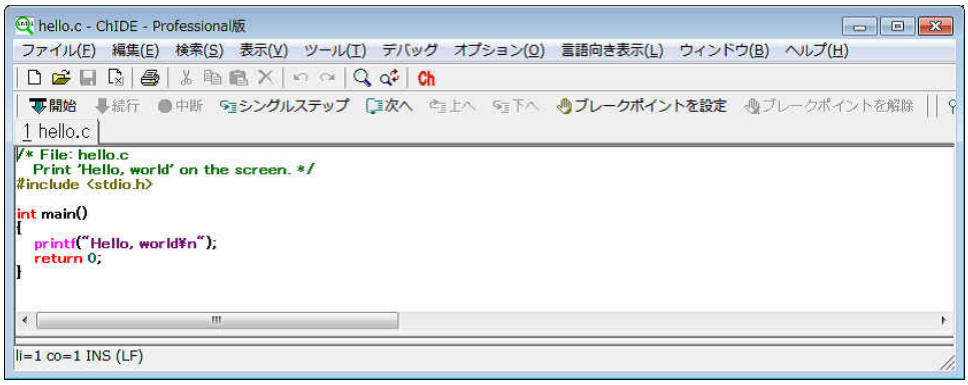


図 3: ChIDE 内に、行番号、余白、折りたたみ表示用余白なしに表示されたプログラム

このドキュメントをコマンド［ファイル］ → ［名前を付けて保存］により、図 4 のように、ファイル名 `hello.c` として保存します。

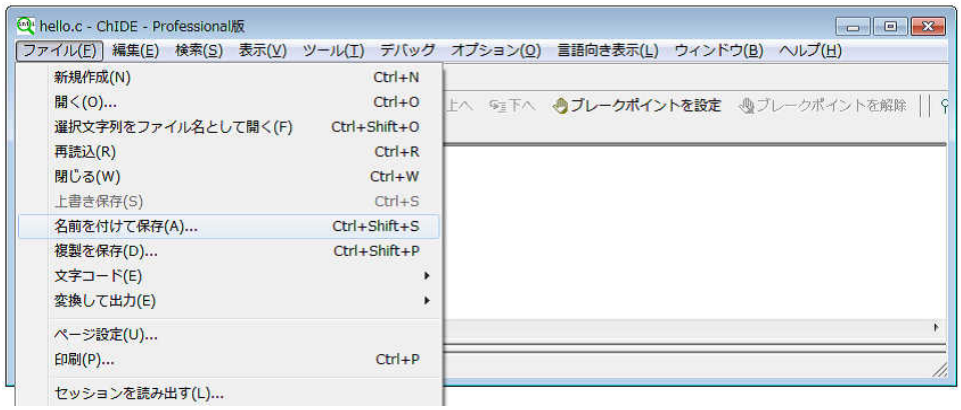


図 4: ChIDE で編集したプログラムの保存

デバッグバーの下にあるそのファイル名のタブを右クリックして、図 5 のように、コマンド［名前を付けて保存］を選択し、そのプログラムを保存することもできます。

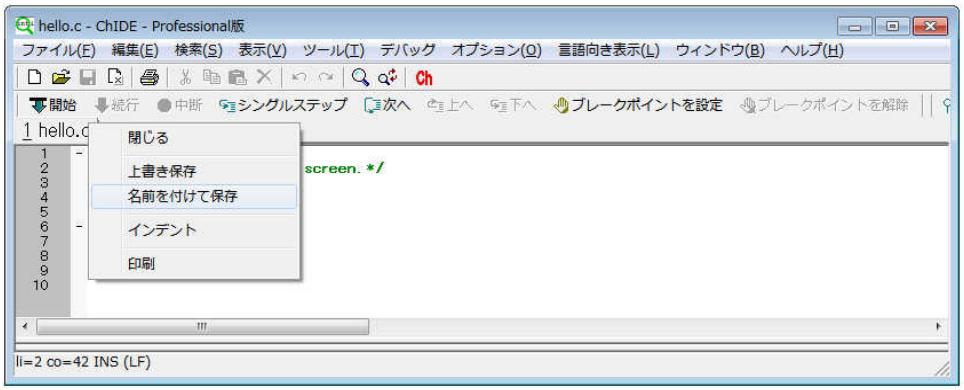


図 5: ファイル名を右クリックして ChIDE で編集したプログラムを保存

このプログラム `hello.c` は `CHHOME/demos/bin/hello.c` に含まれていて、［ファイル］ → ［開く］コマンドを用いてロードすることも可能です。ここで、`CHHOME` は Ch のホームディレクトリで、Windows 上では、Ch のホームディレクトリは既定で C ドライブ上の `C:/Ch` になります。Windows 上では、Windows エクスプローラの一覧にあるプログラムは、これをドラッグして ChIDE 上にドロップし、その編集ウィンドウ内で開くことも可能です。

ファイル拡張子 `.c`、`.ch`、`.cpp`、`.cc`、および `.cxx` をもつか、またはファイル拡張子をもたない C/Ch/C++ プログラムは、ChIDE ですぐに実行可能です。

プログラム `hello.c` を実行するには、図 6 のように、［実行］ボタンをクリックするか、または［ツール］ → ［実行］を選択します。

［実行］または［ツール］ → ［実行］コマンドを選択する代わりに、ファンクションキー `F2` を押しても、プログラムを実行することが可能です。

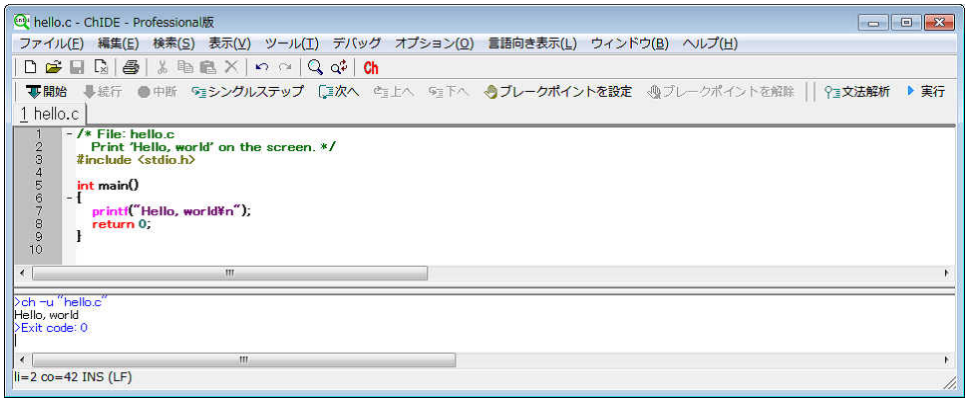


図 6: ChIDE 上の編集ウィンドウ内のプログラムの実行とその出力

ChIDE には、編集ウィンドウ、デバッグウィンドウ、デバッグコマンドウィンドウ、およびアウトプットウィンドウの 4 つのウィンドウがあります。デバッグウィンドウは、編集ウィンドウの下または右に配置されています。このウィンドウのサイズは起動時にはゼロになっていますが、編集ウィンドウとの境界をドラッグすることにより、拡大することができます。デバッグコマンドウィンドウはデバッグウィンドウの下または右に配置されています。同様に、アウトプットウィンドウもデバッグウィンドウの下または右に配置されています。アウトプットウィンドウはデバッグコマンドウィンドウの左に配置されています。このウィンドウは起動時にはゼロになっていますが、デバッグウィンドウとの境界をドラッグすることにより、拡大することができます。

既定では、プログラムからの出力は、アウトプットウィンドウ内に指定されています。

［表示］ → ［左右に分割］コマンドは、ChIDE の水平方向のレイアウト変更に用いられます。ここでは、編集ウィンドウを左に、デバッグウィンドウを真ん中に、そしてアウトプットおよびデバッグコマンドウィンドウを右側に配置されます。現在のセッションにおける、ChIDE の位置とサイズ、編集ウィンドウ、デバッグウィンドウ、およびアウトプットウィンドウのサイズは、ChIDE が閉じられる際に保存されます。ChIDE が次回起動される際には、前回のセッションでこれらの保存された値が、新しいセッションに対して使用されます。コマンド［表示］ → ［既定のレイアウト］は、ChIDE のグローバルおよびユーザーオプションファイル内の値を使用し、ChIDE がこの既定値を使うようにリセットします。`CHHOME/demos/bin/hello.c` にあるものと同じプログラム `hello.c` （ここで、`CHHOME` は Ch 用のホームディレクトリであり、Windows では `C:/Ch` のようになっています）は、［ファイル］ → ［開く］コマンドを使用してロードすることもできます。

プログラム `hello.c` が実行されると、アウトプットウィンドウがまだ表示されていな場合はこれが表示され、以下の実行結果

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

が、図 6 のように表示されます。

ChIDE から出力された 1 行目の青字の行

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

は、このコマンドにより、Ch がプログラム `hello.c` を実行したことを示しています。

黒字の行は、Ch プログラムの実行結果です。ChIDE から出力された最後の青字の行は、プログラムが終了したことを示しています。終了コードが 0 であることは、プログラム中のステートメント

```
return 0;
```

または

```
exit (0);
```

により、実行が正常に終了したことを意味しています。プログラムの実行中に問題が発生した場合、あるいは

```
return -10;
```

または

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

のように、プログラムが return あるいは exit ステートメントで非ゼロの値で終了した場合、終了コードは-1 になります。

ChIDE は、Ch が発生したエラーメッセージを正しく受け取ることができます。このことを確かめるには、プログラムに意図的に誤りを追加します。たとえば、

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

を

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

のように変更します。修正したプログラムに対して［実行］または［ツール］ → ［実行］を選択すると、その結果は以下のようになります。

```
 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'
```

実際の実行画面は、図 7 のようになります。

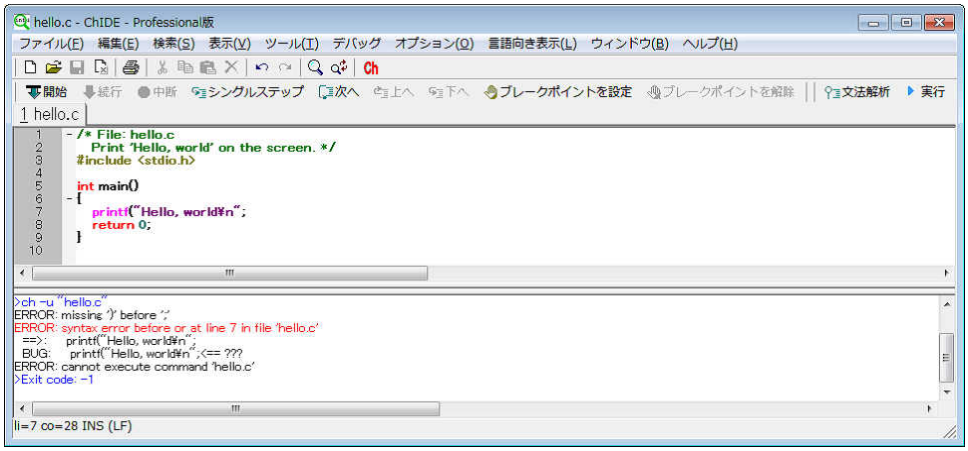


図 7: プログラム hello.c の実行結果の出力におけるエラー行

このプログラムの実行が失敗したため、終了コード-1 が

```
Exit code: -1
```

のように、出力ウィンドウの最後に表示されます。
図 7 のようなアウトプットウィンドウ内の赤字のエラーメッセージをマウスの左ボタンでダブルクリックすると、編集ウィンドウ内の不正な書式とエラーメッセージを含んだ行が図 8 のように黄色の背景で強調表示されます。

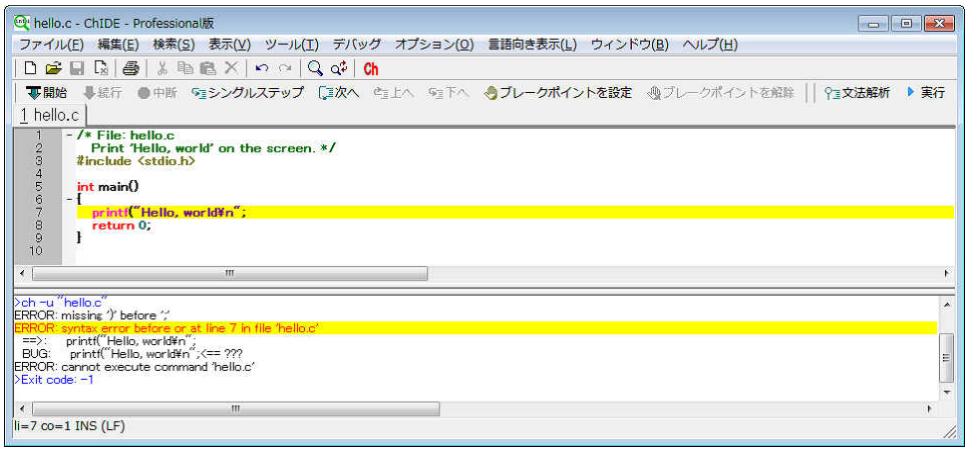


図 8: プログラム hello.c 実行出力からのエラー行の検索

文字カーソルがこの行に移動して、必要な場合には、このウィンドウが自動的にスクロールして、この行が表示されます。この単純な例では問題がどこにあるかを見つけるのは簡単ですが、大きなファイルの場合、[ツール] → [次のエラーメッセージ] コマンドまたはファンクションキー F4 を使用して、レポートされたエラー個所を表示することができます。[ツール] → [次のエラーメッセージ] を実行すると、アウトプットウィンドウ内の最初のエラーメッセージと、編集ウィンドウ内の該当する行が、黄色い背景で強調表示されます。

コマンド [ツール] → [前のエラーメッセージ]、またはファンクションキー Shift+F4 は、直前のエラーメッセージの表示に使用されます。

コマンド [表示] → [アウトプットウィンドウ] を選択すると、アウトプットウィンドウを開く／閉じることができます。コンソールウィンドウの内容は、図 9 のように、[表示] → [アウトプットウィンドウを消去] により消去することができます。

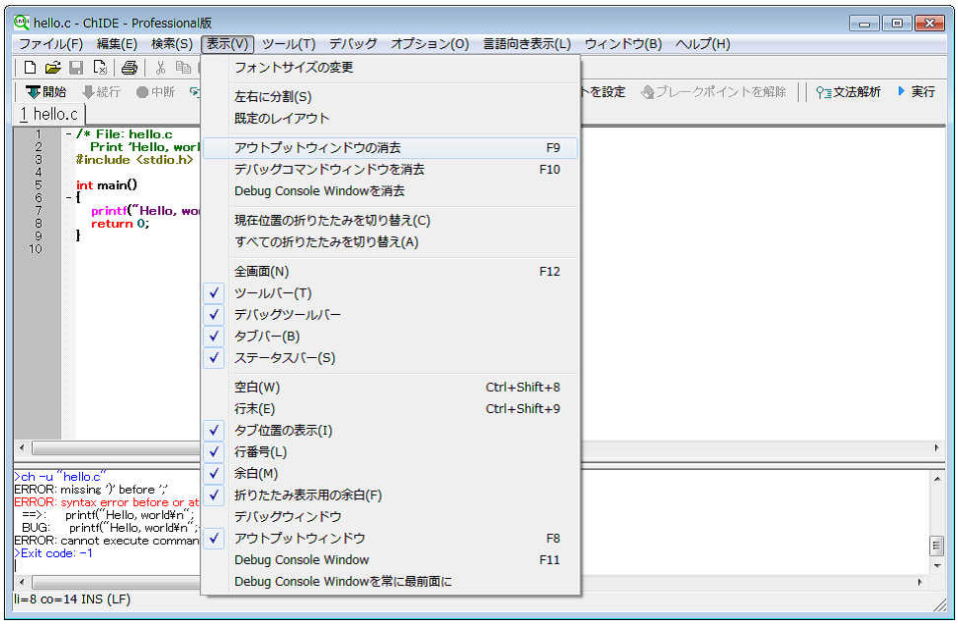


図 9: アウトプットウィンドウの内容の消去

コマンドの実行が失敗し、終了に非常に時間がかかる場合は、[プログラムの強制終了] ボタン、または [ツール] → [実行の停止] コマンドを使用して、プログラムの実行を停止します。

[文法解析] コマンドまたは [ツール] → [文法解析] コマンドを用いて、プログラムを実行せずに書式エラーをチェックすることができます。

読みやすさとソフトウェアのメンテナンスのために、プログラムの各行に適切にインデントが設定されます。ツールバー上のコマンド [ツール] → [インデント] は、編集ウィンドウ内のプログラムに適当にインデントを設定します。デバッグバーの下にあるファイル名のタブを右クリックし、コマンド [インデント] を選択して、プログラムにインデントを設定します。図 5 は、ファイル名が右クリックされたときのコマンド [インデント] を示しています。

ChIDE は、`scanf()` のような C 関数を通したユーザー入力を必要とするプログラムを実行することもできます。ChIDE では、コマンドライン引数を扱うことも可能です。ChIDE を用いて、Ch で C および C++プログラムを実行する際の詳細な情報は、図 10 のように、[ヘルプ] メニューから [ChIDE 解説書] をクリックすることによって、オンラインで得ることができます。

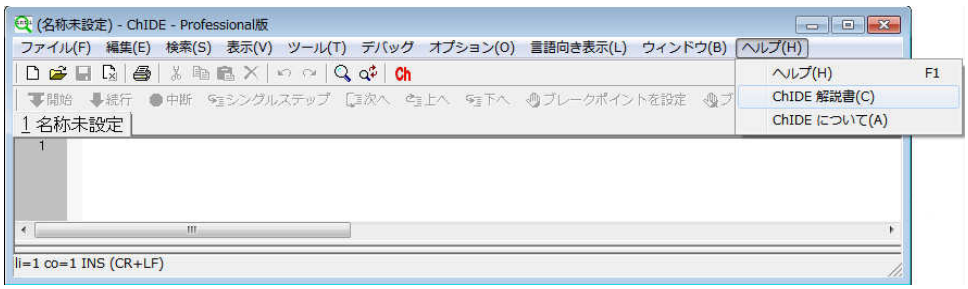


図 10: ChIDE の使用法に関するオンラインヘルプ情報の取得

3　C/Ch/C++プログラムのデバッグ

ChIDE は、バイナリ C プログラム用の典型的なデバッガで利用可能なすべての機能を備えています。「開始」と「シングルステップ」のようなデバッグインターフェースコマンドは、図 11 のようになっています。

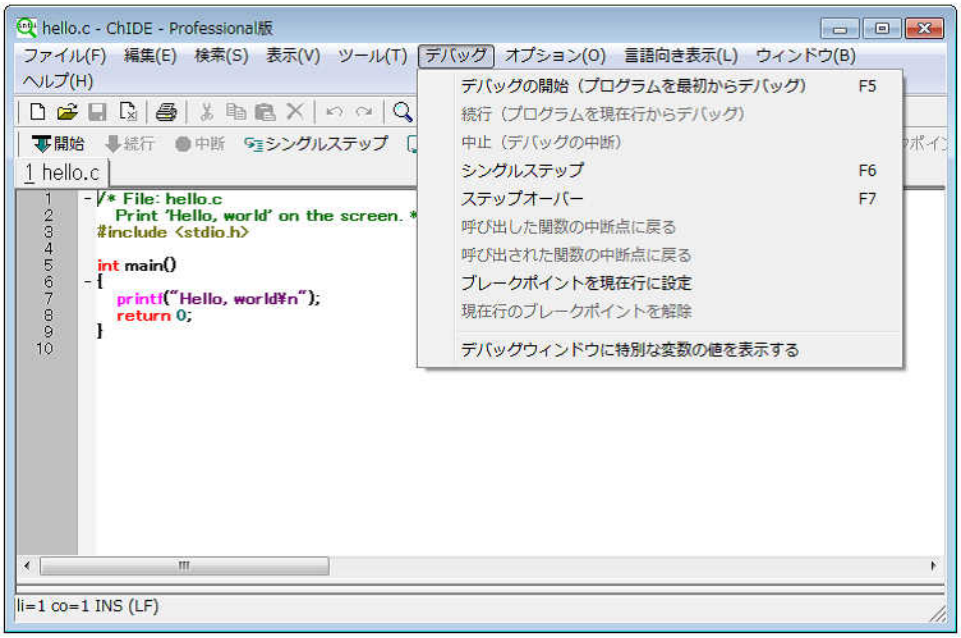


図 11: [デバッグ] メニュー

これらは、図 12 のように、デバッグバー上で直接使用することもできます。

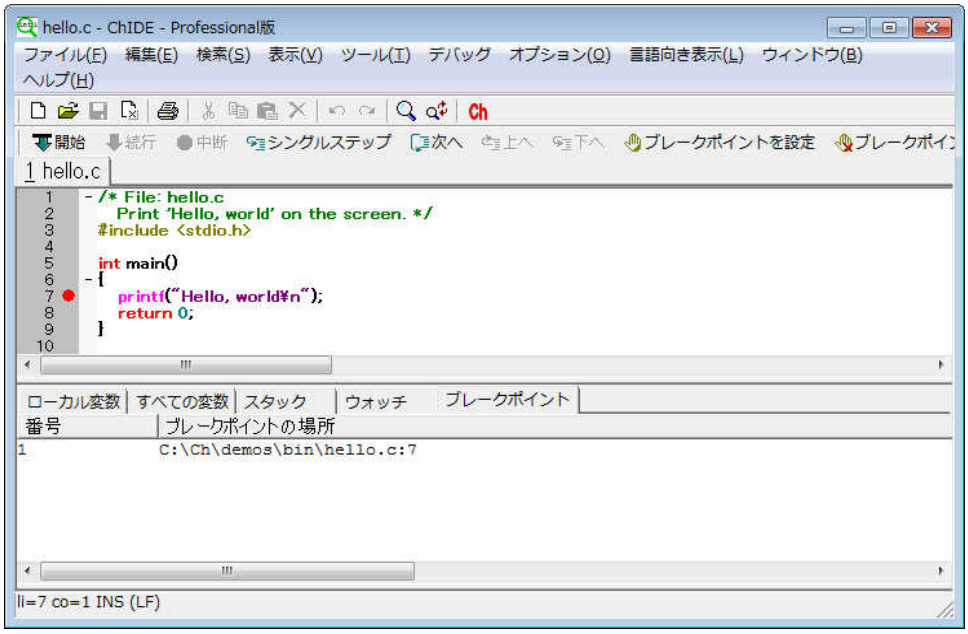


図 12: ブレークポイントの設定

3　C/CH/C++プログラムのデバッグ

デバッグ中の任意の時点で使用可能なデバッグバー上のコマンドは、クリック可能になっています。クリックできないコマンドは、グレーアウトされています。

デバッグモードでは、[開始] コマンドまたはファンクションキー F5 により、編集ウィンドウ内のプログラムを実行することができます。プログラムの実行は、ブレークポイントにヒットすると停止します。プログラムを 1 行ずつ実行するには、[シングルステップ] コマンドまたはファンクションキー F6 を使用します。一方、[ステップオーバー] コマンドまたはファンクションキー F7 は関数をステップオーバーして、次の行を実行対象にします。デバッグ中に [続行] コマンドをクリックすると、プログラムは、ブレークポイントにヒットするか、あるいは最後まで実行されます。

プログラムの実行前またはデバッグ中に、新規のブレークポイントを追加して、これにヒットしたときにプログラムの実行を停止させるようにすることができます。ある行に対するブレークポイントは、図 12 のように、その行の左余白（[表示] → [余白] をクリックして表示させる）をクリックして設定します。このブレークポイントを解除するには、この行の左マージンにある赤い丸印をクリックします。デバッガのブレークポイントは、図 12 のように、デバッグウィンドウの上のデバッグウィンドウ選択バーの [ブレークポイント] タブをクリックして、調べることができます。デバッグウィンドウは、各ブレークポイントに対するブレークポイント番号とその場所を表示します。現在の行に対するブレークポイントは、[ブレークポイントを現在行に設定] コマンドをクリックしても設定可能です。これを解除する方法として、[現在行のブレークポイントを解除] メニューを選択することもできます。ブレークポイントが何も設定されていない場合、コマンド [現在行のブレークポイントを解除] はクリック不可になっています。しかしながら、ブレークポイントは以下のような初期化をともなう宣言文には設定できません。

```
int i = 10;
```

実行中およびデバッグ中のプログラムを編集すべきではありませんが、あえてそうしようとすると、警告メッセージ

> 警告：デバッグ中にこのファイルに加えられた変更は現在のデバッグセッションには反映されません。

が表示され、そのプログラムの実行が終了し、編集可能になります。コードの追加／削除によりプログラムが編集されると、そのプログラムに対するブレークポイント設定が自動的に更新されます。

[シングルステップ] コマンドは、ある関数の中へのステップインに使用されます。その関数がすでにバッファ内にロードされているファイルの中にない場合、その関数を含むファイルがロードされます。プログラムの実行の最後に、デバッグ中にロードされたそのファイルがバッファから削除されます。しかしながら、ブレークポイントがそのロードされたファイルに設定されている場合、プログラムの実行が終了しても、そのファイルはバッファ内にとどまります。

デバッグコマンドウィンドウ内で [debug] コマンドを用いると、あとで説明されているように、ブレークポイントを関数と変数制御に対して設定することもできます。

コマンドの実行が失敗し、終了に非常に時間がかかる場合は、[中止] ボタンを使用して、プログラムの実行を停止します。

デバッグモードでプログラムが実行されると、標準入出力とエラーストリームが、図 13 のように、独立した [Debug Console Window] にリダイレクトされます。

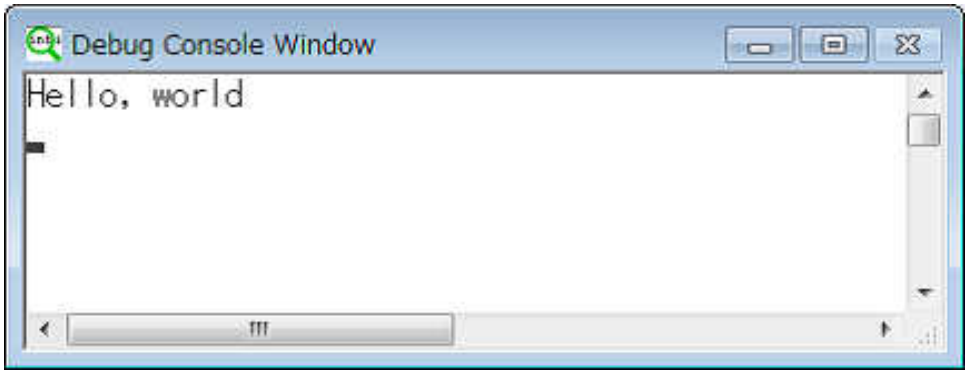


図 13: デバッグ時の入出力用 Debug Console Window

既定では、このコンソールウィンドウは常に他のウィンドウの上に表示されます。この既定の動作は、[表示] → [Debug Console Window を常に最前面に表示] コマンドにより、オン／オフの切り換えを行うことができます。このコンソールウィンドウは、[表示] → [Debug Console Window] コマンドにより、開いたり閉じたりすることができます。このコンソールウィンドウの内容をクリアするには、図 9 のように、[表示] → [Debug Console Window を消去] コマンドを使用します。この「コンソールウィンドウ」のウィンドウおよびフォントサイズはもちろん、背景と文字の色は、このウィンドウの左上隅の ChIDE アイコンを右クリックし、[プロパティ] メニューを選択して変更することができます。なお Windows Vista 上でこのような変更を行う場合、ChIDE を管理者権限で実行する必要があります。

[シングルステップ] または [ステップオーバー] コマンドにより、プログラムが 1 行ずつ実行された場合、現在のスタック内の変数名とその値は、デバッグウィンドウ上で、デバッグウィンドウ選択バーの [ローカル変数] メニューをクリックすることにより、調べることができます。プログラムの実行制御が関数内にあるときに、[locals] コマンドを入力すると、その関数のローカル変数と引数の値が表示されます。プログラムの実行制御がスクリプトの関数内にないときに [locals] コマンドを入力すると、そのプログラムのグローバル変数の値が表示されます。図 14 のように、CHHOME/demos/bin ディレクトリにあるプログラム func.c が 9 行目を実行対象にし、緑色でハイライトされているとき、ローカルの整数 i と n は 1 と 10 に、一方 double 型の配列 a は、1、2、3、4、5 を含んでいます。

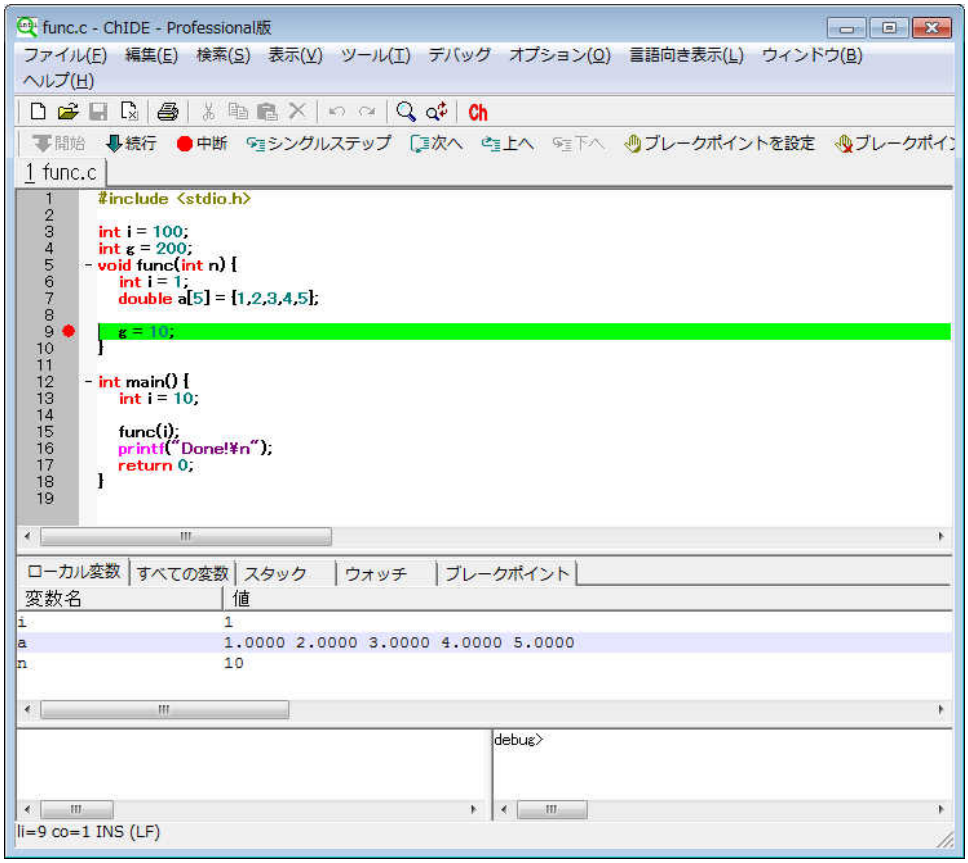


図 14: 現在呼び出されている関数内のローカル変数名と値の表示

デバッグ中に関数スタックを変更することができます。このスタックは、そのスコープ内の変数がデバッグウィンドウ内で表示され、またはデバッグコマンドウィンドウ内でアクセスされるように、呼び出し関数の [上へ] 移動するか、または呼び出された関数の [下へ] 移動することができます。現在の行と呼び出し関数の実行行のハイライト用に異なる色が使用されます。たとえば、図 14 で [上へ] コマンドをクリックすると、プログラムの制御フローは、図 15 で青色にハイライトされているように、呼び出し関数 `main()` の 15 行目に移動します。

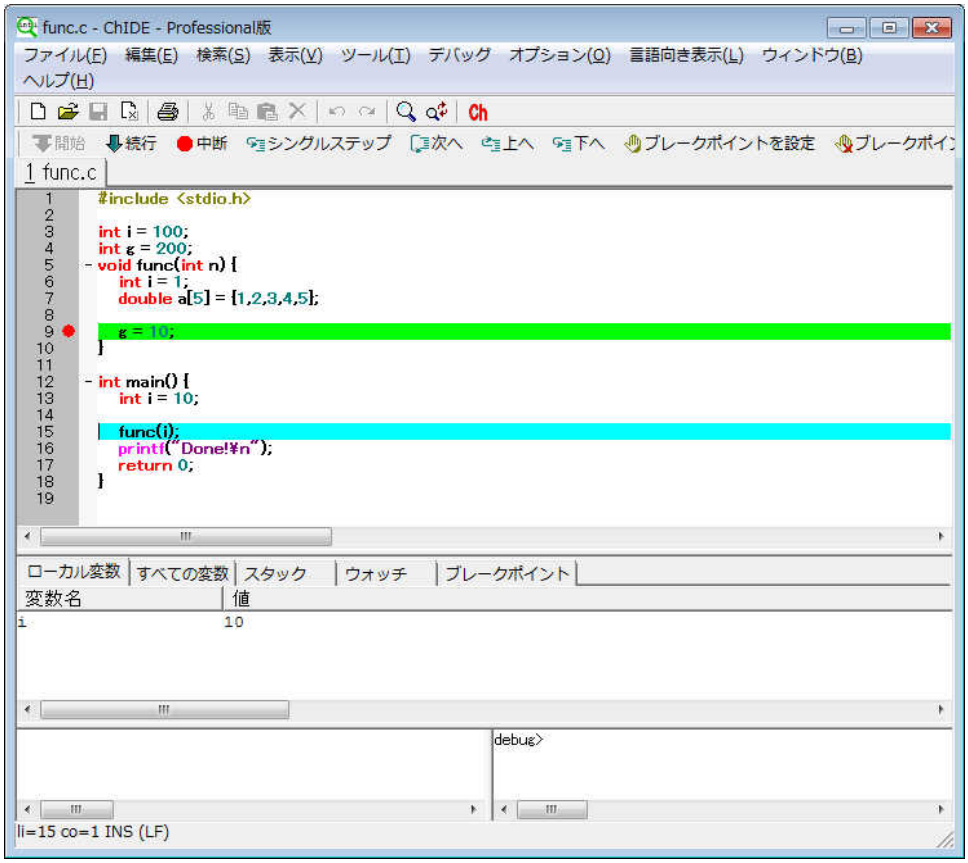


図 15: 呼び出し関数内のローカル変数名と値の表示

図 14 では、[下へ] メニューはクリック不可になっていますが、このメニューは、現在のスタックが上に移動させられた際、図 15 ではクリック可能になっています。この時点でのデバッグウィンドウは、唯一の定まった変数である、呼び出し関数 `main()` 内の変数 `i` の名前と値を表示します。

デバッグウィンドウ上部の [スタック] タブは、各スタック内の、関数、メンバ関数、またはプログラム名および対応するスタックレベルを表示します。現在実行中の関数のスタックレベルは 0 であり、一方レベル n+1 は、スタックレベル n の関数から呼び出されている関数です。

たとえば図 16 に示されているように、関数 `func()` は、プログラム `func.c` により起動された関数 `main()` により、呼び出されます。

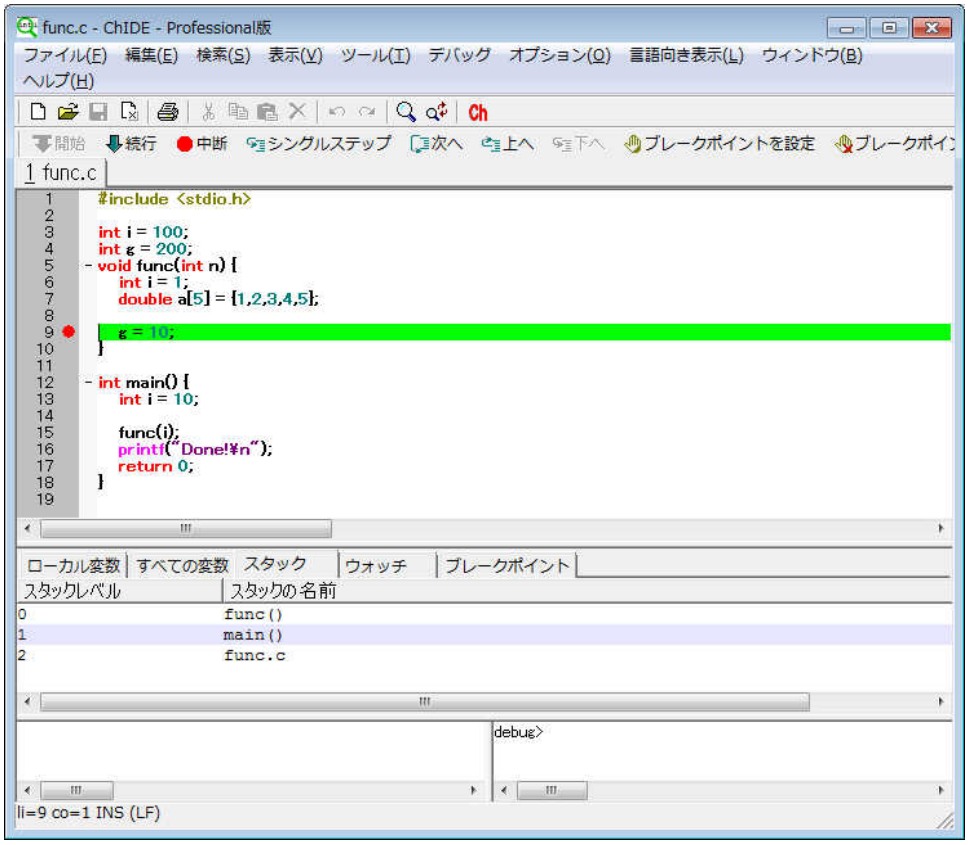


図 16: 実行点に対する異なるスタックの表示

全スタック内の変数名の名前とその値は、デバッグウィンドウ選択バーの［変数］タブで、図 17 のように表示されます。

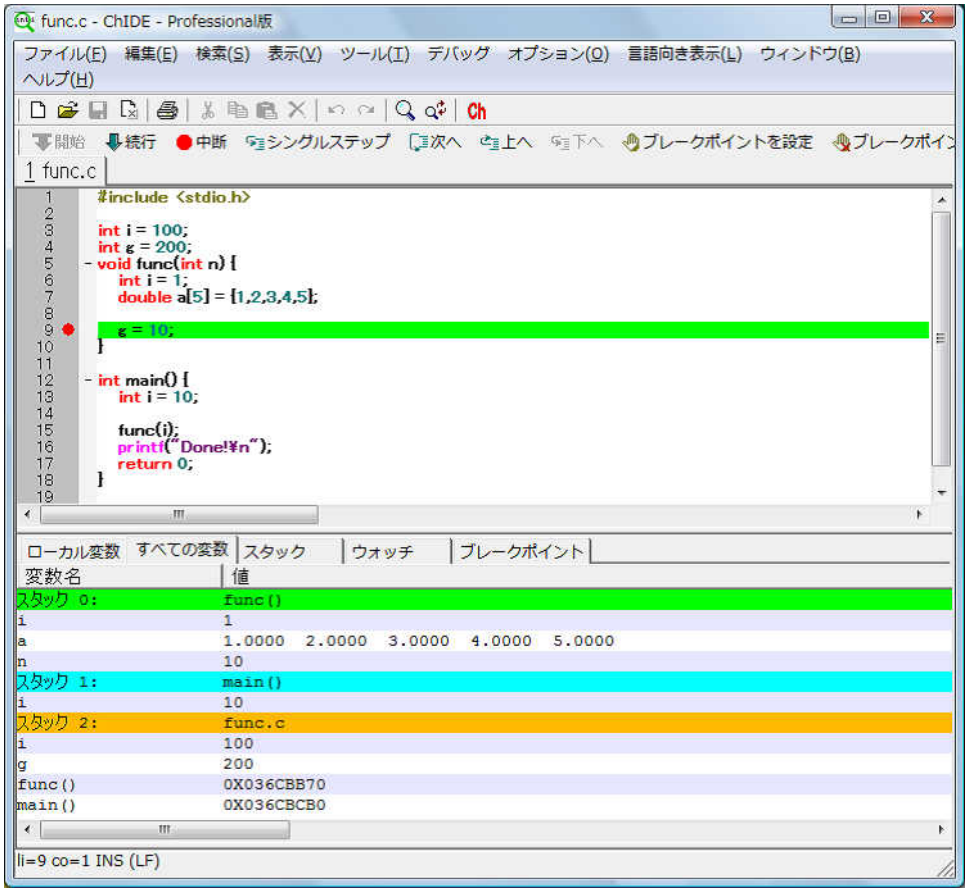


図 17: 全スタック内のすべての変数の名前と値の表示

スタックレベルは、図 15 で示されているように、編集ウィンドウ内の現在の行と呼び出し関数の実行行に対応した色でハイライトされます。図 17 では、プログラムは 9 行目で停止します。関数 `func()` と `main()` 内のグローバル変数はもちろん、ローカル変数の名前と値もデバッグウィンドウ内に表示されます。おわかりのように、9 行目の実行前のグローバル変数 g の値は 200 になっています。

図 11 で表示されている、デバッグメニュー内のコマンド

[デバッグウィンドウに特別な変数の値を表示する]

をクリックすると、`__func__`のような特殊な変数の名前と値がデバッグウィンドウの［ローカル変数］と［変数］タブ内に表示されます。

4　デバッグコマンドウィンドウ内での [debug] コマンドの使用

プログラムのデバッグ中には、多くのデバッグコマンドがデバッグコマンドウィンドウ内で使用可能です。プロンプト

```
debug>
```

がデバッグコマンドウィンドウ内に表示され、デバッガがデバッグコマンドを受け付け可能になったことを示します。[help] コマンドを入力すると、図 18 のように、使用可能なすべてのコマンドが表示されます。

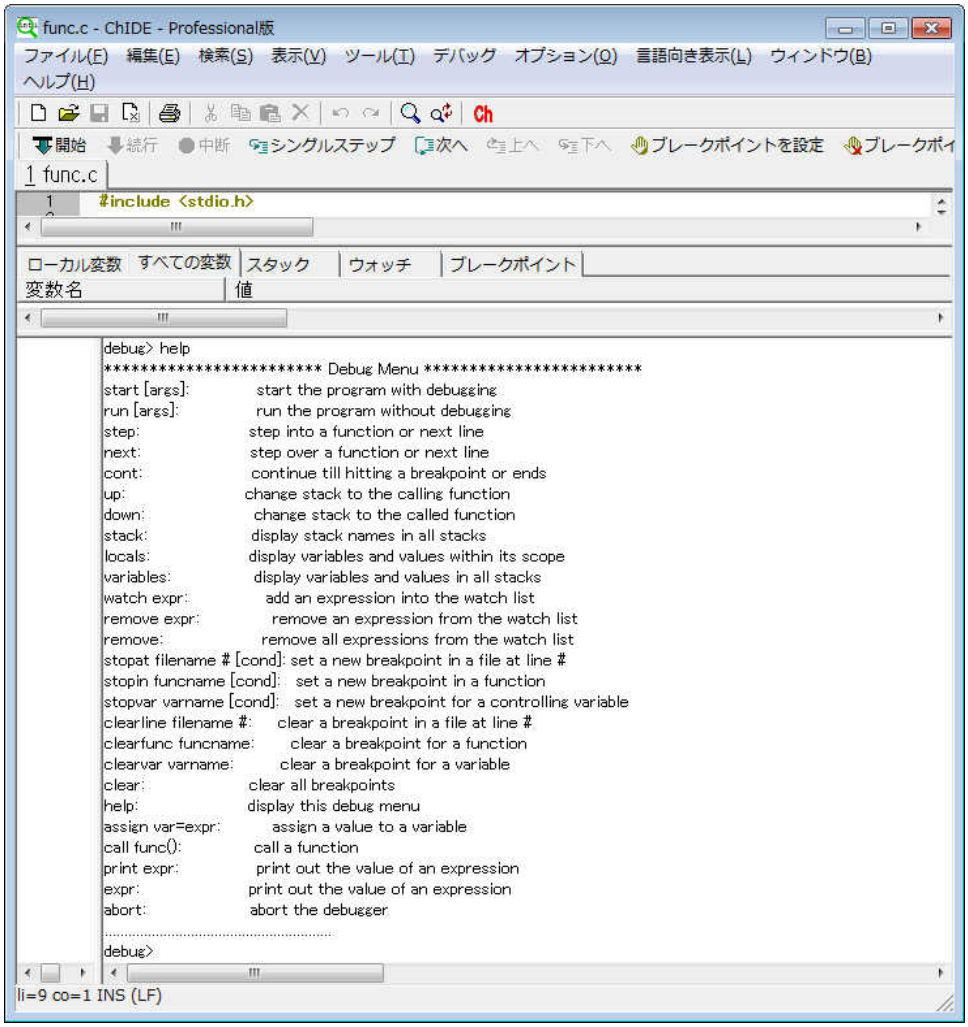


図 18: デバッグコマンドウィンドウ上のデバッグコマンド

コロン（:）の左側のメニューはコマンドを示し、その右側はそのコマンドの動作について説明しています。デバッグコマンドウィンドウ内では、デバッグバー上の全コマンドを使用することができます。さらに、デバッグコマンドウィンドウ内でしか使用できない機能がいくつかあります。

変数、式、および関数は、[assign]、[call]、および [print] コマンドで扱われます。[assign] コマンドは値を変数に割り当て、[call] は関数を起動し、[print] は変数または関数を含む、式

の値を出力します。void 型の式を出力することは、void 型を返す関数の場合も含めて、無効な操作です。式をタイプするだけで、その式の値が表示されます。その式が void の型を返す関数である場合、その関数のみが呼び出されます。たとえば、コマンド

```
debug> assign i=2*10
debug> call func()
debug> print i
20
debug> 2*i
40
debug>
```

は、変数 `i` に値 10 を割り当て、関数 `func()` を呼び出し、変数 `i` がそのスコープ内で有効な場合には、式 `2*i` の値を出力します。他の例として、図 19 のように、プログラム `func.c` が実行され、9 行目で停止した場合、式 `2*g` とともに変数 `a` と `i` の値が、デバッグコマンドウィンドウ上で対応するコマンドを入力することにより、表示されます。

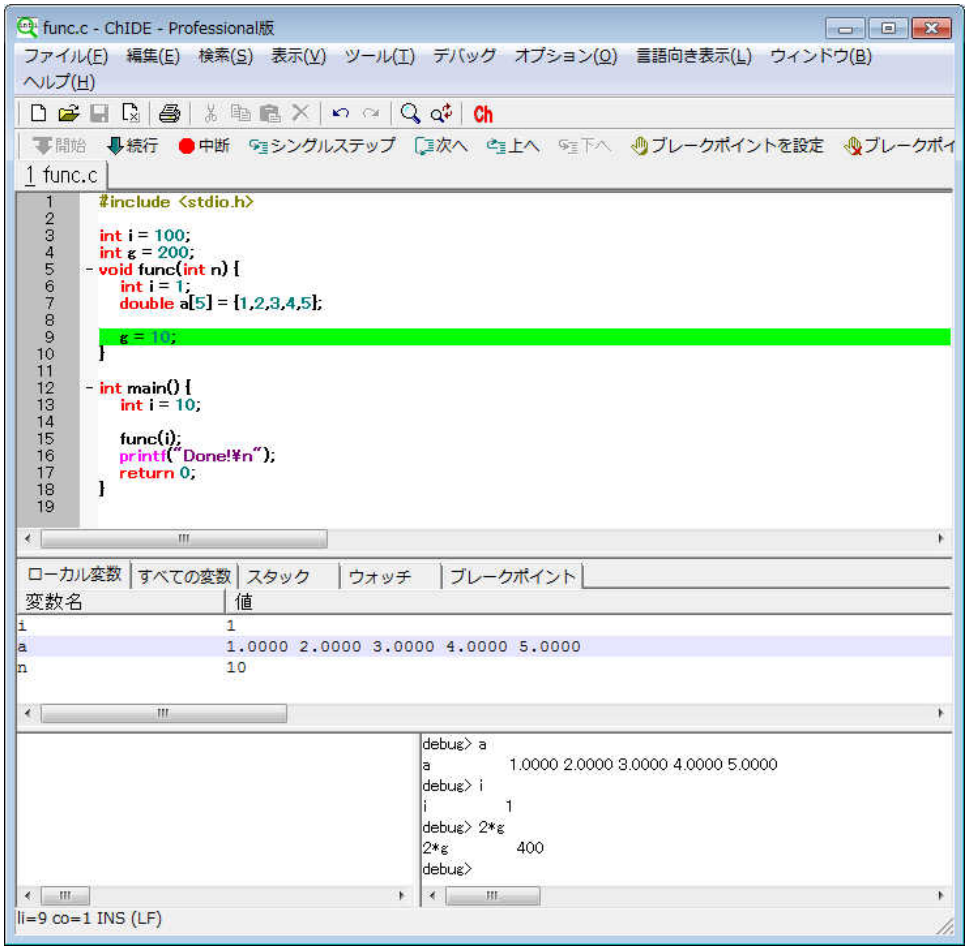


図 19: デバッグコマンドウィンドウ内でのデバッグコマンドの使用

`[start]` コマンドは、プログラムのデバッグを開始します。`[start]` と `[run]` コマンドのオプションのコマンドライン引数は処理されて、`main()` 関数に対する引数に渡されます。たとえば、図 20

4 デバッグコマンドウィンドウ内での *[DEBUG]* コマンドの使用

のプログラム C:\Ch\demos\bin\commandarg.c を実行するため、デバッグコマンド

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

は、文字列 "C:\Ch\demos\bin\commandarg.c"、"-o"、"option1"、"-v"、"option2"、および"option3 with space" を、エレメント argv[0]、argv[1]、argv[2]、argv[3]、argv[4]、および argv[5]、すなわち、Ch スクリプト commandarg.c の main 関数

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

の引数 argv にそれぞれ割り当てます。

空白文字を含んだコマンドライン引数は、上記の文字列 "option3 with space" で示されるように、二重引用符で囲む必要があります。

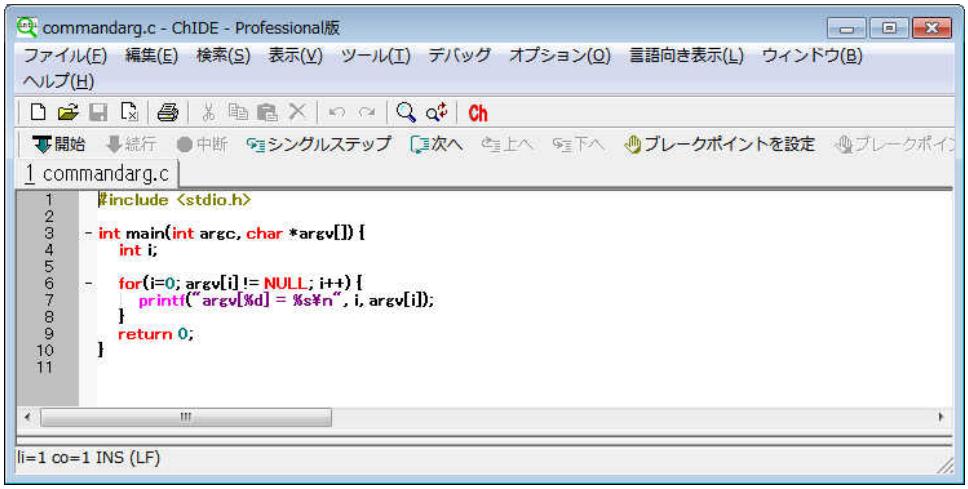


図 20: コマンドライン引数を扱うプログラム

プログラムはブレークポイントにヒットすると停止します。[run] コマンドを入力すると、プログラムは、ブレークポイントを無視し、デバッグされることなしに実行されます。[step] または [next] コマンドにより、デバッグバー上のコマンドと同様に、プログラムを 1 行ずつ実行することができます。[step] コマンドは、関数にステップインし、一方 [next] コマンドは、関数をステップオーバーして次の行を実行対象にします。デバッグ中に [cont] コマンドを起動して、プログラムをブレークポイントにヒットするか、または最後まで実行させることができます。デバッグ中に関数スタックを変更することが可能です。[up] および [down] コマンドにより、関数スタックのスコープ内の変数にデバッグコマンドウィンドウ内でアクセスできるように、それぞれ関数スタックを、その呼び出し関数の上に移動させるか、または呼び出された関数の下に移動させることができます。すべてのスタック内の関数またはプログラム名を表示させるには、[stack] コマンドを使用します。現在のスタック内の変数の名前とその値を表示させるには、[locals] コマンドを使用します。[variables] コマンドを入力すると、各スタック内のスコープにあるすべての変数の名前と値が表示されます。

[watch] コマンドは、シングル変数を含む式をウォッチ式のリストに追加します。ウォッチ式は、プログラムの実行前にも実行中にも追加することが可能です。[remove expr] コマンドを使用すると、ウォッチ式のリストから式を削除することができます。また、[remove] コマンドは、ウォッチ式リスト内のすべての式を削除します。たとえば、デバッグコマンドウィンドウ内で以下のコマンド

4 デバッグコマンドウィンドウ内での *[DEBUG]* コマンドの使用

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

を実行すると、図 21 のように、式 `2*g` と変数 `i` がウォッチ式のリストに追加されます。 プログラムがブレークポイントで停止するか、または次のステートメントにステップインすると、デバッグウィンドウ選択バーの［ウォッチ］タブをクリックすることにより、図 21 のように、これらのウォッチ式の値がデバッグウィンドウ内に表示されます。

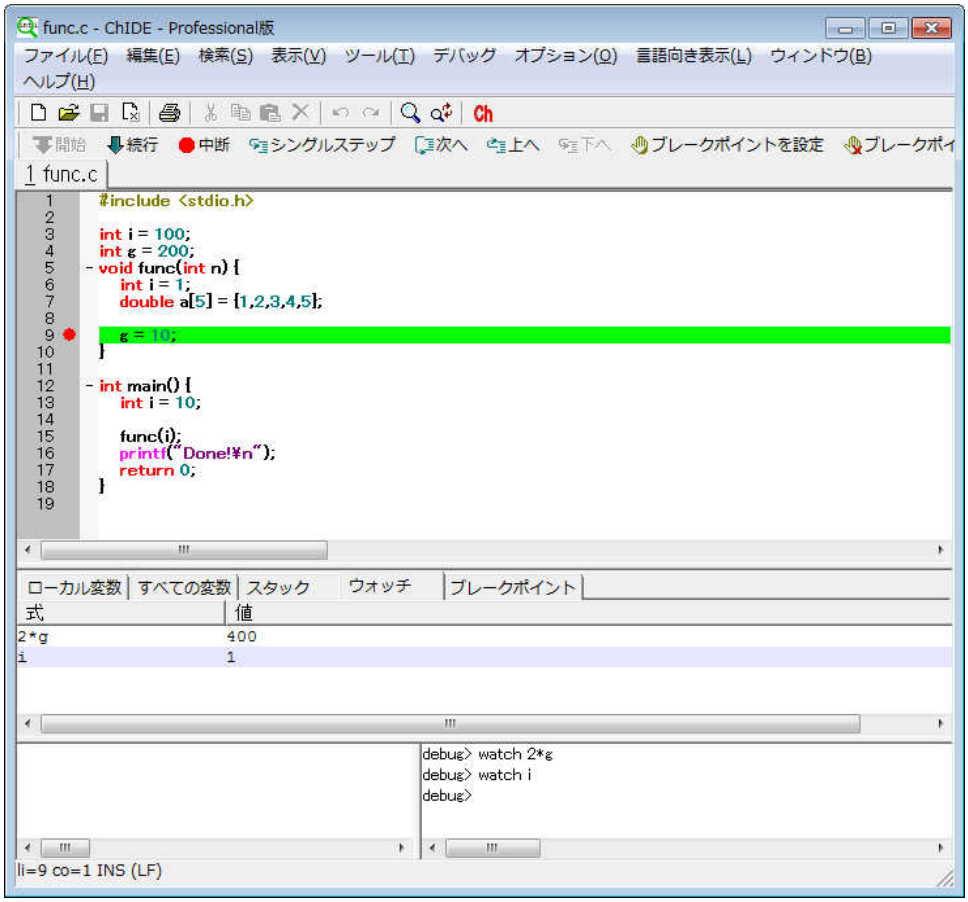


図 21: デバッグコマンドウィンドウ内でウォッチ式と変数を設定し、その値をデバッグウィンドウ内に表示

プログラム実行前またはデバッグ中に、新規のブレークポイントを追加して、プログラムの実行を停止させることができます。ブレークポイントは、次の 3 項目の指定、すなわち、ファイル名と行番号、関数、および制御変数、により設定されます。ブレークポイントが関数内に設定されると、プログラムは関数の最初の実行可能な行で停止します。ブレークポイントが変数に対して設定されると、プログラムは、その変数値が変更されたときに停止します。各ブレークポイントには、オプションの条件式があります。ブレークポイントの場所に到達すると、条件式があればその条件式が評価されます。ブレークポイントは、式（ブレークポイント設定時に指定する必要があります）が真かまたは変更された場合のみ、ヒットします。 既定では、ブレークポイントは、式が真の場合のみヒットします。`[stopat]` コマンドは、引数としてファイル名と行番号を指定することにより、新規のブレークポイントを設定します。プログラムがこの場所に到達すると、実行が停止されます。`[stopin]` コマ

ンドは、関数に対して新規のブレークポイントを設定します。プログラムがこの関数の最初の実行可能行に到達すると、実行が停止されます。[stopvar] コマンドは、制御変数に対して新規のブレークポイントを設定します。この変数は、プログラム実行中に評価されます。プログラムは、この変数の値が変更されると、実行を停止します。これらのコマンドのいずれかが起動されると、ブレークポイントがブレークポイントリストに追加されます。各ブレークポイントに対するオプションの条件式とトリガ手段は、これらのコマンドの最後の 2 つの引数として渡されます。たとえば、ブレークポイントを完全なパスと行番号を用いて、ファイル内に設定する場合は、以下のようになります。

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

ブレークポイントの場所に到達すると、オプション式 [condexpr] が評価されます。引数 [condtrue] が真であるか、またはない場合、ブレークポイントは、この式の値が真である場合にヒットします。そうでない場合、ブレークポイントは、この式の値が変更された場合にヒットします。たとえば、コマンド

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

は、C:/Ch/demos/bin ディレクトリにあるファイル func.c 内の 6 行目にブレークポイントを設定します。コマンド

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

も、ファイル func.c の 6 行目にブレークポイントを設定しますが、次のような違いがあります。すなわち、ファイル func.c の 6 行目の位置のブレークポイントに到達すると、式 i+j が評価され、この式の値が真のときに、ブレークポイントがヒットします。上記の式は、以下の式と同等です。

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

コマンド

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

もやはりファイル func.c の 6 行目にブレークポイントを設定します。この場合、ファイル func.c の 6 行目の位置のブレークポイントに到達すると、式 i+j が評価され、この式の値が変更されたときに、ブレークポイントがヒットします。逆に、適切な引数をともなって、コマンド [clearline]、[clearfunc]、および [clearvar] を実行すると、リスト内のそれぞれ、行、関数、および変数型が削除されます。[clear] コマンドは、デバッガ内のすべてのブレークポイントを削除します。

プログラムの実行が失敗し、終了に時間がかかり過ぎる場合は、[abort] コマンドを用いて、プログラムを停止させることができます。デバッグコマンドウィンドウは、図 9 で表示されているように、[表示] → [デバッグコマンドウィンドウを消去] をクリックすることにより、消去できます。

5　Ch コマンドシェル入門

Ch は、コマンドで操作を行うコマンドシェルとして使用することも可能です。MS-DOS シェル、Bash-shell、または C-shell などの他の一般的なシェルのように、Ch シェル内でコマンドが実行可能で

す。しかしこれらの一般的シェルとは異なり、Ch シェルでは、C および C++における式、ステートメント、関数、プログラムが、容易に実行、参照可能です。したがって、Ch コマンドシェルは、C/C++学習に対する理想的なソリューションです。教官は、Ch をノート PC とともに授業で対話的に用いて、特に学生の質問に答える際に、迅速にプログラミングの要点を説明することができます。学習者は、コンパイル／リンク／実行／デバッグの退屈なサイクルを抜きにして、C/C++の様々な特徴をすばやく学ぶことも可能です。初心者の学習を援助するため、Ch はエラー発生時に、**セグメンテーションフォールト**および**バスエラー**または「クラッシュ」などの不可解でわかりにくいメッセージの代わりに、数多くのわかりやすい警告とエラーメッセージ[2]を発生するようになっています。

Ch シェルは、ch コマンドを実行して起動します。Windows および Mac OS X x86 上では、Ch コマンドシェルは、デスクトップまたは ChIDE のツールバー上にある、図 22 のような、赤色の **Ch** アイコンをクリックして簡単に起動することもできます。

図 22: Windows および Mac OS X x86 デスクトップ上の Ch アイコン

Ch シェルが Windows 上で起動されたあと、既定でユーザーアカウントが管理者であると仮定すると、シェルウィンドウの画面プロンプト

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

が表示されます。ここで、`C:/Documents and Settings/Administrator` は、図 23 のようなデスクトップ上のユーザーの**ホームディレクトリ**です。シェルウィンドウのウィンドウとフォントサイズはもちろん、そのテキストと背景の色も、ウィンドウの左上隅の Ch アイコンを右クリックし、[プロパティ] メニューを選択して変更することができます。なお、Windows Vista 上で ChIDE にこのような変更を行うには、管理者権限が必要となります。表示されるディレクトリ `C:/Documents and Settings/Administrator` は、**カレント（現在の）作業ディレクトリ**とも呼ばれます。ユーザーアカウントが管理者でない場合、アカウント名 *Administrator* が、適切なユーザーアカウント名に変更される必要があります。

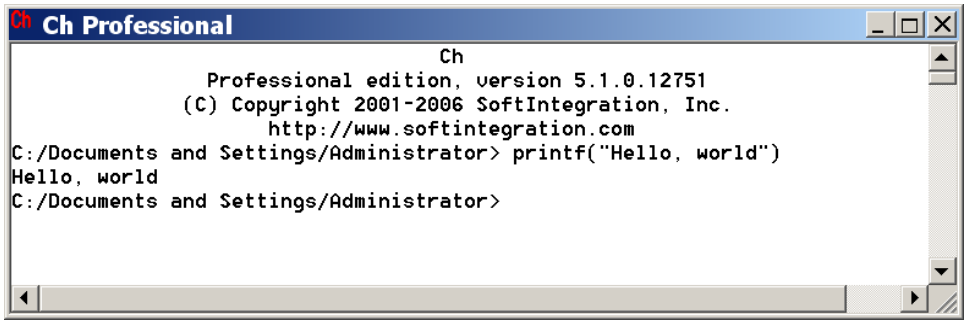


図 23: Ch コマンドシェル

プロンプトは、システムが Ch シェルであり、ユーザーのターミナルキーボード入力の受け入れが可能であることを示しています。Ch シェルの既定のプロンプトは、以下のように構築されています。

[2]ただし現行バージョンでは、これらのメッセージはすべて英語となっています。

入力内容が文法的に正しい場合、正常に実行されます。実行終了後、システムプロンプト> が再度表示されます。プログラムまたは式の実行中にエラーが発生した場合、Ch シェルは該当するエラーメッセージを出力し、ユーザーによるプログラムのデバッグを援助します。

C のすべてのステートメントと式は、Ch コマンドシェル内で対話的に実行されます。たとえば、出力 `Hello, world` は、関数 **printf()** を呼び出すことによって以下のように対話的に得られ、

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

図 23 のようになります。図 23 と比較すると、本書の説明のスペースを節約するために、最後のプロンプト `C:/Documents and Settings/Administrator>`が省略されています。なお、コマンドモード上で C プログラムの該当するステートメントを実行する際には、ステートメントの最後のセミコロンは省略可能です。上の実行例では、関数 **printf** の呼び出し時のセミコロンはありません。

5.1 ファイルハンドリング用のポータブルコマンド

システムプロンプト>において、C プログラムとステートメントだけでなく、（カレント作業ディレクトリを表示する **pwd** のような）他の任意のコマンドも実行可能です。このシナリオでは、Ch は Windows 上の MS-DOS シェルと同様に、コマンドシェルとして使用されます。

コマンドは、Ch コマンドシェルまたは Ch プログラム内で実行可能です。数百ものコマンドとそのオンラインドキュメントがシステムに含まれています。これらのすべてを知っている人はいません。各コンピュータウィザードは、常に使用できる作業ツールを備えていて、そこに何があるかという漠然としたアイデアを提供しています。このセクションでは、表 1 に掲載されているファイルハンドリング用のもっともありふれたコマンドの使用法について、例を挙げて説明します。

表 1: ファイルハンドリング用のポータブルコマンド

コマンド	用法	説明
cd	cd	ホームディレクトリを変更
	cd *dir*	ディレクトリ *dir* に変更
cp	cp *file1 file2*	*file1* を *file2* にコピー
ls	ls	作業ディレクトリ中の内容の一覧
mkdir	mkdir *dir*	新規のディレクトリ *dir* を作成
pwd	pwd	作業ディレクトリの名前を出力（表示）
rm	rm *file*	*file* を削除
chmod	chmod +x *file*	*file* のモードを変更して、実行可能にする
chide	chide *file.c*	*file.c* の編集と実行のために、ChIDE を起動する

Ch シェル上で実行されるこれらのコマンドは、Windows、Linux または Mac OS X のような異なるプラットフォームに渡ってポータブルであることを再度強調するしておく必要があります。これらのコマンドを使用すると、ユーザーは C プログラムを実行するために、システム上のファイルを効果的に操作することができます。

Ch が Windows 上で、既定のディレクトリ C:/Ch にインストールされていると仮定します。カレント作業ディレクトリは、C:/Documents and Settings/Administrator であり、これはユーザーのホームディレクトリでもあります。ファイルハンドリング用ポータブルコマンドのアプリケーションは、以下のように、Ch シェルでのコマンドの対話的実行を例として説明されます。

```
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
```

表 1 の「**用法**」に示されているように、**mkdir** コマンドは、作成するディレクトリとして 1 個の引数を取ります。最初にコマンド

```
mkdir c99
```

を用いて、c99 というディレクトリを作成します。次に、コマンド

```
cd c99
```

を用いて、この新規に作成したディレクトリ C:/Documents and Settings/Administrator/c99 に変更します。さらに、コマンド

```
pwd
```

でカレント作業ディレクトリを表示します。図 2 のような、ディレクトリ C:/Ch/demos/bin にある C プログラム hello.c は、コマンド

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

を用いて、この作業ディレクトリに同じ名前でコピーされます。コマンド

```
ls
```

により、カレントディレクトリにあるファイルの一覧が表示されます。この時点では、ディレクトリ C:/Documents and Settings/Administrator/c99 にあるファイルは、hello.c のみです。すべてのプログラムをあとで容易に探すことができるように、作成した全 C プログラムをこのディレクトリに保存することをお勧めします。

最後に、コマンド

```
chide hello.c
```

により、プログラム hello.c が起動され、図 2 に表示されているように、ChIDE で編集と実行が可能になります。学級でのプレゼンテーションに対して、複数のソースファイルを以下のように、1 つのコマンドで開くことは、ときには便利です。

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

空白文字を含むパスを扱うコマンドを使用するには、そのパスを二重引用符で囲む必要があります。たとえば、ファイル hello.c を削除するには、以下のように指定します。

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

5.2 プログラムの対話的実行

C プログラムをコンパイルせずに Ch シェルで対話的に実行することは、非常に簡単です。たとえば、前のセクションのように、C:/Documents and Settings/Administrator/c99 がカレント作業ディレクトリであると仮定します。このディレクトリ内のプログラム hello.c は Ch で実行され、以下のように、Hello, world の出力が得られます。

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

Ch コマンドシェル内のプログラム実行結果の終了コードは、システム変数 **_status** に保存されます。プログラム hello.c が正常に実行されたので、**_status** がコマンドライン入力されると、上のように終了コードは 0 になります。

Unix では、C プログラム hello.c をコマンドとして簡単に使用するには、このファイルが実行可能でなければなりません。コマンド **chmod** は、ファイルのモードを変更します。以下のコマンド

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

は、プログラム hello.c を実行可能にし、Ch コマンドシェル内で実行できるようにします。

5.3 Ch コマンドのパス設定と検索

コマンドを実行するために、これをコマンドシェルのプロンプトに入力すると、コマンドシェルはそのコマンドを指定済みのディレクトリ内で検索します。Ch シェルでは、文字列型のシステム変数 **_path** が、検索されるコマンドのディレクトリを含んでいます。各ディレクトリは、文字列 **_path** 内で、セミコロンにより区切られています。Ch コマンドシェルが起動される際、システム変数 **_path** はいくつかの既定の検索パスを含んでいます。たとえば、Windows 上での既定の検索パスは以下のとおりです。

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

ユーザーは、スタートアップファイル（これについては、あとで説明されます）内の文字列関数 **stradd()** を使用して、コマンドシェル用の検索パスへの新規のディレクトリを追加することができます。この関数は、文字列型の引数を追加して、これを新規の文字列として返します。たとえば、ディレクトリ C:/Documents and Settings/Administrator/c99 はコマンドの検索パスの中にはありません。カレントの作業ディレクトリが
C:/Documents and Settings/Administrator であるときに、このディレクトリ内のプログラム hello.c を起動しようとすると、Ch シェルはこのプログラムを見つけることができず、以下のようなエラーメッセージを出力します。

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

Ch が起動されるか、または Ch プログラムが実行されると、既定では、ユーザーのホームディレクトリ上にスタートアップファイル**.chrc**（Unix 環境）または**_chrc**（Windows 環境）が存在する場合には、それが実行されます。本書ではこれ以降、Ch が Windows 環境で使用され、スタートアップファイル**_chrc** がユーザーのホームディレクトリ上にあると仮定します。このスタートアップファイルは多くの場合、コマンド、関数、ヘッダファイルなどの検索パスを設定します。Windows 上では、既定の設定のスタートアップファイル**_chrc** は、Ch のインストール時に、ユーザーのホームディレクトリ上に作成されます。しかしながら、Unix 上では、既定でユーザーのホームディレクトリ上には、スタートアップファイルはありません。システム管理者は、このようなスタートアップファイルをユーザーのホームディレクトリに追加することができます。スタートアップファイルがまだホームディレクトリ上にない場合、以下のように、Ch を -d オプション付きで起動し、

```
ch -d
```

サンプルのスタートアップファイルを、ディレクトリ `CHHOME/config/` からユーザーのホームディレクトリにコピーすることができます。ここで、`CHHOME` は文字列`"CHHOME"`ではなくて、Ch がインストールされたシステムファイルパスであることに注意してください。たとえば、既定で Ch が `C:/Ch`（Windows）または`/usr/local/ch`（Unix）にインストールされた場合、Windows 上では、以下の Ch シェルのコマンド

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

はスタートアップファイル**_chrc** をユーザーのホームディレクトリ `C:/Documents and Settings/Administrator` に作成します。このローカルの Ch 初期化スタートアップファイル_chrc は、図 24 のように、ChIDE エディタにより、検索パスの編集用に開くことができます。

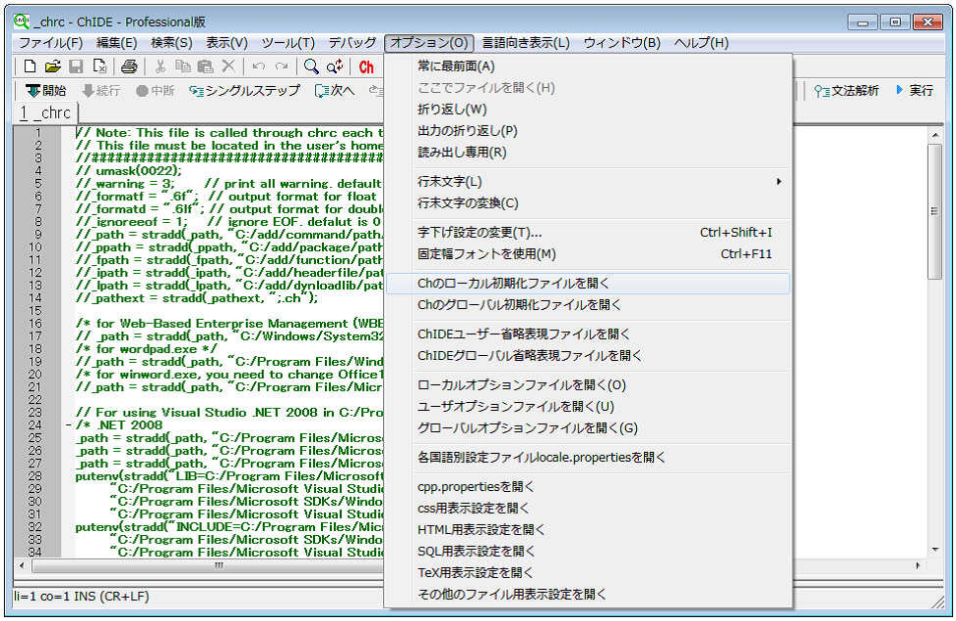


図 24: ローカルの Ch 初期化スタートアップファイルを編集用に開く

Linux では、上記のコマンド `ch -d` は、デスクトップ上に Ch アイコンも作成します。Ch が ChIDE とともにインストールされる場合、ChIDE 用のアイコンもデスクトップ上に作成されます。

ディレクトリ C:/Documents and Settings/Administrator/c99 をコマンド検索パスに含ませるには、以下のステートメント

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

をユーザーホームディレクトリ上のスタートアップファイル **_chrc** に追加して、このディレクトリ上のコマンド hello.c が、カレントの作業ディレクトリの設定に関係なく、起動できるようにする必要があります。ディレクトリ C:/Documents and Settings/Administrator/c99 が検索パス **_path** に追加されたあとでは、Ch コマンドシェルを再起動する必要があります。このようにして、プログラム hello.c をこのディレクトリ上で、以下のように実行することができるようになります。

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

コマンドに対する **_path** と同様に、Ch のヘッダファイルは、システム変数 **_ipath** で指定されるディレクトリ上で検索されます。各パスもまた、セミコロンで区切られます。たとえば、ステートメント

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

は、ディレクトリ C:/Documents and Setting/Administrator/c99 を、

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

のようなプロプロセッサディレクティブ include によりインクルードされているヘッダファイル用の検索パスに追加します。また、このディレクトリをステートメント

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

により、関数ファイル用の検索パスに追加することもできます。関数ファイルには、関数定義が含まれています。

Unix 上では、コマンドの検索パスは既定で、カレント作業ディレクトリを含みません。コマンドの検索パスにカレント作業ディレクトリを含ませるには、以下のステートメント

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

をユーザーのホームディレクトリ上のスタートアップファイル **.chrc** に追加する必要があります。関数呼び出し stradd(_path, ".;") は、'.' で表されたカレントディレクトリをシステム検索パス **_path** に追加します。

5.4 式とステートメントの対話的実行

話を簡単にするため、本書ではこれ以降、Ch コマンドシェル内で表示されるプロンプトは、>のみであるとします。C 式がコマンドシェル内で入力されると、これは Ch で評価され、その結果が画面上に表示されます。たとえば、

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

のように、式 1+3*2 が入力されると、出力が 7 となります。

有効な C 式は、すべて Ch シェル内で評価されます。したがって、Ch は計算機として使用すると便利です。

別の例として、以下のように、変数を宣言してこれをこのあとに続く計算で使用することができます。

```
> 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
```

上記の C ステートメントでは、変数 i が 4 バイトの int 型として、宣言されています。次に、i に対する整数値 30 が、十進、16 進、2 進数で表示されています。異なる数値表記の整数定数も、上記のように、変数 i に割り当てることが可能です。最後に、負数 −2 に対する 2 の補数表記も表示されています。C における他のすべてのデータ型の特性もまた、対話的に表現可能です。関数 **fopen**() でオープンされたファイルストリームを使用して、入力関数 **fscanf**() と出力関数 **fprintf**() ファミリ用の異なるフォーマットの指定子も、このようにして試すことができます。

既定では、float または double 型の値は、それぞれ小数点以下 2 桁または 4 桁で表示されます。たとえば、以下のとおりです。

```
> float f = 10
> 2*f
20.00
> double d = 10
> d
10.00000
```

以下のように、すべての C 演算子が対話的に使用可能です。

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

ポインタと変数のアドレスは、以下のように表されます。

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

```
20
> i
20
```

この例では、int へのポインタの変数 p は、変数 i をポイントします。次の例では、以下のように配列とポインタの関係が示されています。

```
> 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
```

式 a[1]、*(a+1)、*p、および p[0] は、すべて同じ要素を参照しています。多次元配列もまた、対話的に扱うことができます。潜在的なバグ検出のため、Ch は配列の境界をチェックします。たとえば、以下のとおりです。

```
> 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
```

5 要素の配列 a で許されるインデックスは、0 から 4 までです。配列 s はヌル文字を含む 5 文字のみを保持することができます。Ch はこのように、既存の C コード内の、配列境界オーバーランに関係したバグを検知することができます。

C の構造体または C++のクラスのアラインメントも、以下のようにチェック可能です。

```
> struct tag {int i; double d;} s
> s.i =20
20
> s
```

```
.i = 20
.d = 0.0000
> sizeof(s)
16
```

この例では、int と double のサイズはそれぞれ 4 および 8 であるにも関わらず、int と double 型の 2 つのフィールドをもつ構造体 s のサイズは、アラインメントを適正化するために、12 ではなくて 16 になっています。

5.5 関数の対話的実行

プログラムを多くの個別のファイルに分割することができます。各ファイルは、プログラムの任意の場所からアクセス可能な、多くの関連した関数群から成ります。C 標準ライブラリのすべての関数が対話的に実行可能であり、ユーザー定義関数内で使用可能です。たとえば、以下のとおりです。

```
> 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
```

乱数発生関数 rand() は、srand(time(NULL) における時間値を用いて起動の準備がされています。型が特定されない数学関数 sin() を呼び出している関数 add() がプロンプト上に定義され、使用されています。

複数の関数定義を含んでいるファイルは、通常それ自身を Ch プログラムの一部として識別するために、.ch のサフィックスをもっています。Ch プログラミング環境内に関数ファイルを作成することができます。Ch における**関数ファイル** とは、1 つの関数定義のみを含んだファイルのことです。関数ファイル名は、addition.chf のように、.chf で終わります。関数ファイル内の関数ファイル名と関数定義は同一でなければなりません。関数ファイルを用いて定義された関数は、あたかも Ch におけるシステム内蔵関数として扱われます。

コマンドに対する **_path** と同様に、関数は、関数ファイル用のシステム変数 **_fpath** における検索パスに基づいて検索されます。各パスは、セミコロンで区切られています。既定では、変数 **_fpath** はパス lib/libc、lib/libch、lib/libopt、および libch/numeric を Ch のホームディレクトリ内に含んでいます。システム変数 **_fpath** が Ch シェル内で対話的に変更されると、これは、現在のシェル内で対話的に起動された関数に対してのみ、影響を及ぼします。動作中のスクリプトに対しては、現在のシェル内の関数検索パスの設定は、サブシェル内では使用されず、そして継承されません。この場合、システム変数 **_fpath** は、ユーザーのホームディレクトリにあるスタートアップファイル **_chrc**（Windows）または **.chrc**（Unix）内で修正することができます。

たとえば、addition.chf という名前のファイルがプログラム 1 のようなプログラムを含んでいる場合、関数 addition() は、入力された 2 つの引数 a と b の和 $a+b$ を計算するために呼び出されるシステム内蔵関数として扱われます。

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

プログラム 1: 関数ファイル addition.chf

関数ファイル addition.chf が
C:/Documents and Settings/Administrator/c99/addition.chf にあり、ディレクトリ
C:/Documents and Settings/Administrator/c99 が、ステートメント

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

をもった、ユーザーのホームディレクトリにあるスタートアップファイル**.chrc**（Unix）または**_fpath**（Windows）内の関数検索パスに追加される必要があると仮定します。関数 addition() は、

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

のように、コマンドモードで対話的に使用されるか、またはプログラム内部で使用されるかのどちらかです。プログラム 2 において、関数 addition() は、関数ファイル addition.chf の内部で定義されたプロトタイプ関数が起動されるように、main() 関数内で、関数プロトタイプなしに呼び出されます。

```
/* 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;
}
```

プログラム 2: 関数ファイル addition.chf を使用したプログラム

プログラム 2 の出力は、c = 5 となります。関数ファイルの検索パスが適切に設定されていない場合、関数 addition() が呼び出された際に、警告メッセージ

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

が表示されます。
関数が Ch シェル内で対話的に呼び出されると、その関数ファイルがロードされます。関数が呼び出されたあとにその関数ファイルを変更すると、コマンドモードにおけるそれ以降の呼び出しは、依然として、古いバージョンのロード済みの関数定義を使用します。修正したバージョンの新たな関数ファイルを起動するには、コマンド **remvar** に続いて関数名を指定して、システム内の関数定義を削除するか、またはプロンプト上で ch を入力して、新たな Ch シェルを起動するかのいずれかを行います。たとえば、コマンド

```
> remvar addition
```

は、関数 addition() に対する定義を削除します。コマンド remvar は、宣言された変数の削除にも使用されます。

5.6 C++プログラミング機能の対話的実行

Ch では、C プログラムが実行可能であるだけでなく、クラスといくつかの C++機能も、C++コードの対話的実行例として以下に示されているように、Ch でサポートされています。

```
> 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
```

入力と出力は、C++における cin および cout を用いてハンドリングされています。public メソッド tagc::set() は private メンバ m_i をセットし、一方、public メソッド tagc::get() はその値を取得しています。メソッド tagc::get() の引数は、参照渡しされています。クラス tagc のサイズは、メンバ関数用のメモリを除いて、4 バイトになっています。

6　アウトプットウィンドウ内でのコマンドの対話的実行

バイナリコマンドまたは C/C++プログラムは、図 25 に示されるように、アウトプットウィンドウ内部でも対話的に実行することができます。

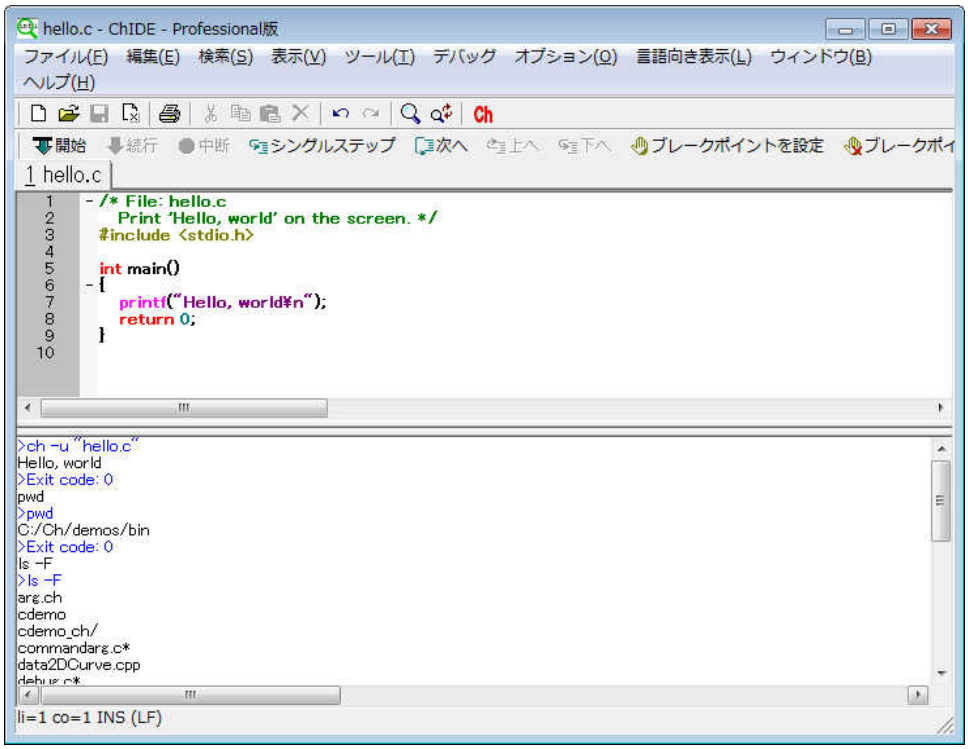


図 25: アウトプットウィンドウ内でのコマンドの実行

6 アウトプットウィンドウ内でのコマンドの対話的実行

図 25 において、プログラム `hello.c` が最初にアウトプットウィンドウ内で実行され、次にコマンド **pwd** がカレントワーキングディレクトリを表示します。コマンド **ls** は、カレントワーキングディレクトリ内のファイルとディレクトリの一覧を表示します。

コマンドのオプションも使用可能です。たとえば、コマンド **ls** は

```
ls -F
```

の書式で実行可能であり、ディレクトリを最後にスラッシュ付きで一覧表示します。

空白文字を含む完全なパスをともなうコマンドを使用するには、以下のように、そのパスを二重引用符で囲む必要があります。

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

7　C/C++プログラムのコンパイルとリンク

ChIDE は、編集ウィンドウ内の編集済みの C/C++プログラムを、C および C++コンパイラを用いてコンパイル、リンクすることもできます。既定では、ChIDE は、Windows 上にインストールされている最新の Microsoft Visual Studio .NET を C および C++プログラムのコンパイルに使用します。Visual Studio コンパイラ用の環境変数とコマンドは、ユーザのホームディレクトリにあり、図 24 のメニューで開いて編集できる、別個のスタートアップ構築ファイル_chrc 内で修正可能です。Linux および Mac OS X x86 上では、ChIDE は、C および C++プログラムのコンパイルに、それぞれ gcc および g++コンパイラを使用します。既定のコンパイラは、[オプション] メニュー内のコマンドで開くことができる C/Ch/C++プロパティファイル cpp.properties を編集することにより変更可能です。

図 26 で表示されている、［ツール］ → ［コンパイル］コマンドは、プログラムのコンパイルに使用することができます。

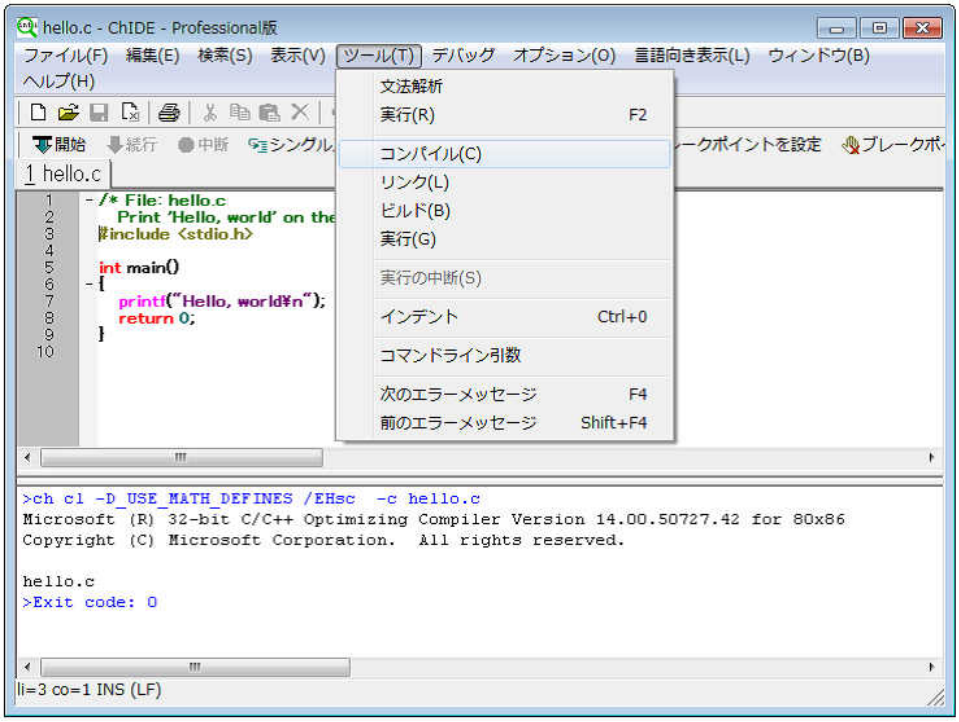


図 26: C/C++プログラムのコンパイル

C または C++プログラムのコンパイル時に発生したエラーメッセージは、ChIDE のアウトプットウィンドウ内に表示されます。Windows 上では、プログラムをコンパイルすると、拡張子.obj をもったオブジェクトファイルが生成されます。このオブジェクトファイルは、［ツール］ → ［リンク］コマンドを用いてリンクされ、実行可能なプログラムが作成されます。Windows 上で実行可能なプログラムは、拡張子.exe をもちます。

カレントディレクトリ上で make ファイル makefile または Makefile が使用可能であれば、［ツール］ → ［ビルド］コマンドを実行することにより、その make ファイル起動されて、アプリケーションがビルドされます。

make ファイルは、ファイルタブ上のファイル名を右クリックし、図 27 のように、コマンド［make］（Linux または Mac の場合）、またはコマンド［make］あるいは［nmake］（Windows の場合）をクリックすることによっても起動することができます。

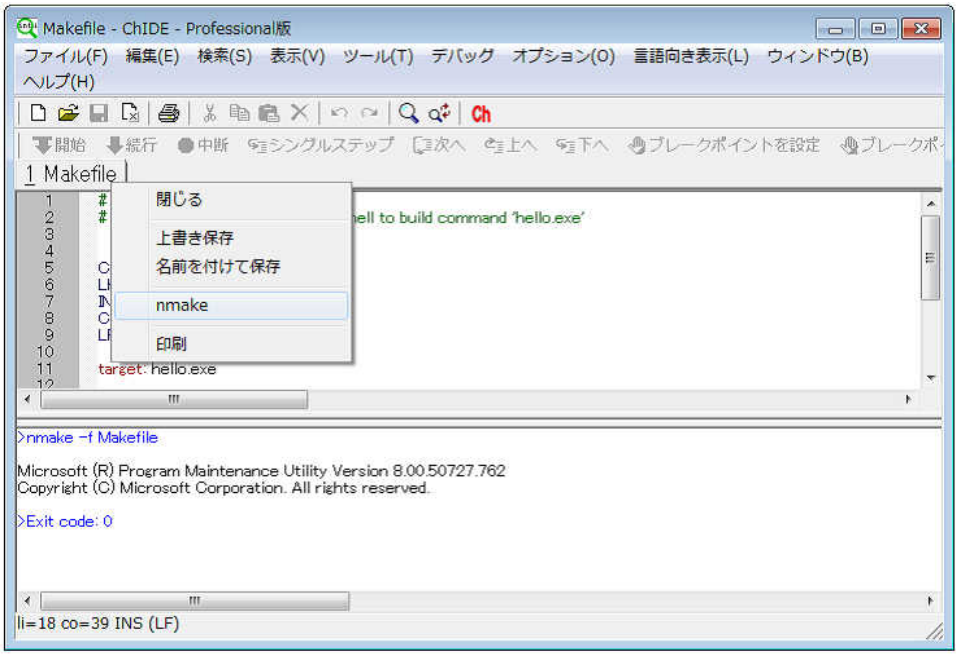


図 27: C/C++ プログラムをコンパイルするための make ファイルの使用

ChIDE が make ファイルの編集に使用される際には、その書式がハイライト表示されます。タブ文字は、ある make コマンドに対してはコマンドを開始する特殊文字として予約されていますので、この文字は保護され、空白文字では置き換えられません。ファイル拡張子 .mak をもつファイル、または以下の名前のファイルは、ChIDE では make ファイルとみなされます。

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

［ツール］ → ［実行］コマンドを実行することにより、作成された実行可能プログラムが実行されます。

8　ChIDE で共通に使用されるキーボードコマンド

ChIDE におけるキーボードコマンドは、共通の Windows および GTK+規則にほとんど準拠しています。すべての移動キー（矢印、page up/down、home、end）に対して、Shift キーを押した際のストリーム選択、および Shift と Alt キーを押した際の長方形領域選択の切り換えが可能です。メニューコマンドのキーボードショートカットは、メニュー内に記載されています。表 2 は、よく使用されるコマンドとそれに対応するキーボードコマンドの一覧を示しています。

8 CHIDE で共通に使用されるキーボードコマンド

表 2: ChIDE で共通に使用されるコマンドとそれに対応するキーボードコマンド

コマンド	キーボードコマンド
ヘルプ情報の参照	F1
Ch 上で C/Ch/C++プログラムを実行	F2
次の検索文字列	F3
前の検索文字列	Shift+F3
次のエラーメッセージ	F4
前のエラーメッセージ	Shift+F4
（プログラムのデバッグを）開始	F5
シングルステップ実行	F6
次のステートメントをステップオーバーして実行	F7
アウトプットウィンドウを閉じる／開く	F8
アウトプットウィンドウを消去	F9
デバッグコマンドウィンドウを閉じる／開く	F10
Debug Console Window を閉じる／開く	F11
全画面表示	F12

索引