開発ツールを使って学ぶ!
C言語プログラミング

坂井弘亮 著
Hiroaki Sakai

開発体験。
プロが使うツールでプログラミング・マスター!

Linux操作・エディタ
diff/patchによるパッチ作成
gitによるコード管理
GDBデバッガ

基本からアルゴリズム入門
分割コンパイル
アーカイブ配布まで!

書籍サポートサイトより、学習用開発環境をダウンロード提供

マイナビ

開発ツールを使って学ぶ!
C言語プログラミング

坂井弘亮 著
Hiroaki Sakai

マイナビ

本書学習環境のダウンロードサービスについて
本書の学習内容を再現できるVM（仮想マシン・PC：Linuxディストリビューション CentOS）イメージファイルのダウンロードサービスを以下のサイトで提供します。
http://kozos.jp/vmimage/progtool.html

また以下の本書サポートサイトでも同じファイルを提供します。2つのサイトで提供するファイルはまったく同じものです。
http://book.mynavi.jp/support/pc/5149/

・サイズは1.2GBあります。インストールし活用することを考えると、お使いのパソコンの空き容量が5GB以上あることを確認の上、ご利用ください。
・ダウンロードしたVMイメージファイルはVirtualBox と VMware Player上で動作します。これらのソフトウェアを別途入手してください。
・VirtualBox と VMware Playerの入手については本書6ページと1章「1.3 VMを使ってみよう」を参考にしてください。
・サイトは予告なく閉鎖することがあります。予めご了承ください。

はじめに

本書はC言語プログラミングと，プログラミングに利用される各種のツール類の「体験書」です．

プログラミングもツールの利用も，その修得には「習うより慣れろ」的な部分が多くあります．このため本書ではチュートリアル的に，体験してみることをテーマとしています．とりあえずひととおりの手順を，手を動かして，試して，体験してみようというわけです．プログラミングの体験に合わせて，各種の開発ツール類も使ってみることで，開発を行う際の実際の手順をひととおり知ることができるかと思います．

本書で体験するのは，C言語による簡単なプログラミング，シェルによるCUIの操作，テキストエディタの操作，ソースコード管理ツールの利用，作業の自動化，スクリプト言語の利用，ソースコードのアーカイブなど，盛りだくさんです．C言語プログラミングとツール類の体験を，交互に進めていきます．

また初心者でも気軽に入門することができるように，必要ツールをインストール済みのＶＭ(Virtual Machine)のイメージを配布しており，体験する環境には，それをそのまま利用できます．

C言語はフリーソフトウェア開発やＯＳ開発，組込み分野などで広く利用されているプログラミング言語です．また昨今，セキュリティ分野などではコンピュータの低いレイヤーの学習が見直されており，そのためにC言語を学ぶ，もしくはC言語を通じて低レイヤーを学ぶといったこともあるようです．

そのような目的で，企業の新人教育などでも他のプログラミング言語と並行して，あえてC言語が扱われたりするという話を聞くこともあります．これらのことを考えても，C言語に触れて体験しておくことの意義は十分にあると言えるでしょう．

しかし実際にプログラミングを行おうとすると，プログラミング言語だけでなく様々なツールの使いかたも覚える必要があることに気がつきます．シェルによるコマンド操作，makeというコマンドによる自動化，gitというツールによるソースコードの管理，デバッガによる動作の確認などです．プログラミングの演習を行う際に，それ以前にシェルによるコマンド操作が立ちいかず演習が進まない，ということも多くあるように思います．

これらはC言語のプログラミングとはまた別の枠として学習することも可能です．しかしプログラミングに利用するツール類なのですから，なるべくならば早い段階に，プログラミングの学習と同時進行で習得したほうが理解もしやすいように思います．

このため本書では，C言語プログラミングの説明と並行して，そうしたツール類の使いかたも体

験していきます．このように同時進行でチュートリアル的に一通りの作業を体験することで，読者の方々にとってより実践的な経験が得られればと思います．

逆に，C言語や各種ツールについて，体系立てて説明するということはしていません．またそれぞれの説明はそれほど深いものではなく「広く浅く」扱っています．もしかしたらその点を，物足りなく感じる読者のかたもいるかもしれません．

本書が提供するのはあくまで「体験」と「きっかけ」です．本格的な修得に関しては，別の書籍などに譲ることにします．本書をぜひ，様々なツールの使いかたを体験し，別の情報源を調べるためのきっかけとして活用していただければと思います．

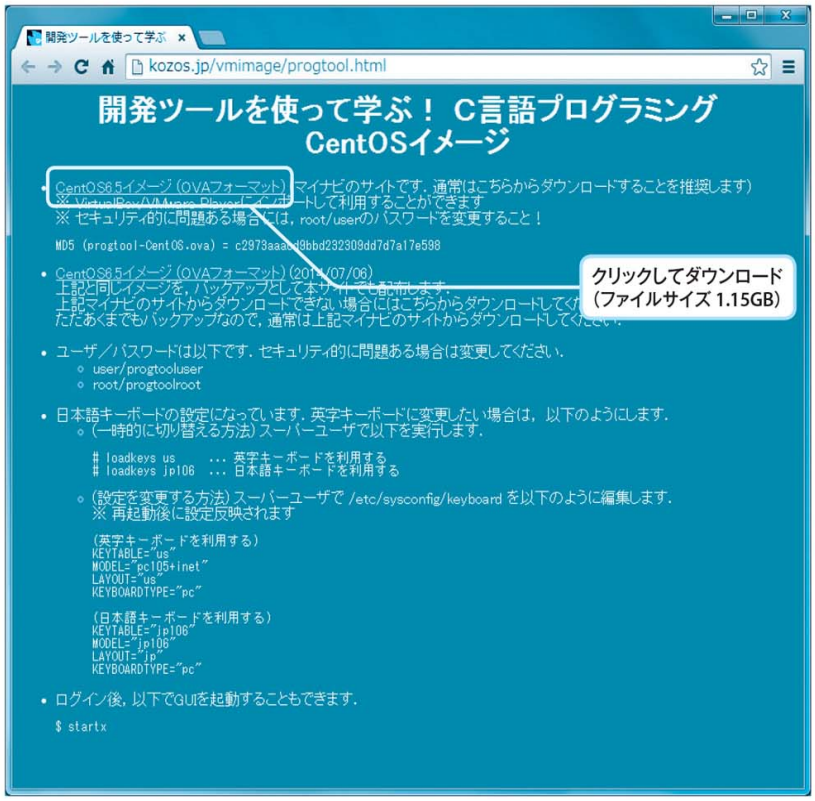
また「体験」は，マウスでアイコンをクリックするようないわゆる「GUI」ではなく，キーボードからのコマンド入力による「CUI」によって行います．これは「GUI」より「CUI」のほうが流行に流されにくく，つぶしが効く知識となるためです．サーバ操作などにも利用できるため，知っておいて損は無い知識と言えます．

自転車の乗りかたは，説明書を読むことではなく実際に乗ってみることで身につくものです．
そしてときには転んでみることも，大切なことでしょう．頭でっかちにならずに，そしてエラーを恐れずに，まずはコマンドを叩いてみましょう．そのような体験を通して得られるものは，意外に大きなものです．

●学習用開発環境のダウンロードについて

本書の学習はLinux（CentOS）上に構築した開発環境にて行いますが，著者が作成したCentOSのVM仮想マシン・PCのイメージファイルをインターネット上で配布します．必要な開発ツール類もすべてインストール済みですので本書の内容をそのまま試すことができます．

http://kozos.jp/vmimage/progtool.html
http://book.mynavi.jp/support/pc/5149/



ダウンロードする「progtool-CentOS.ova」ファイルは1.2GB近くありますので，インストールし活用することを考えると，お使いのPCの空き容量が5GB以上あることを確認の上，ご利用ください．

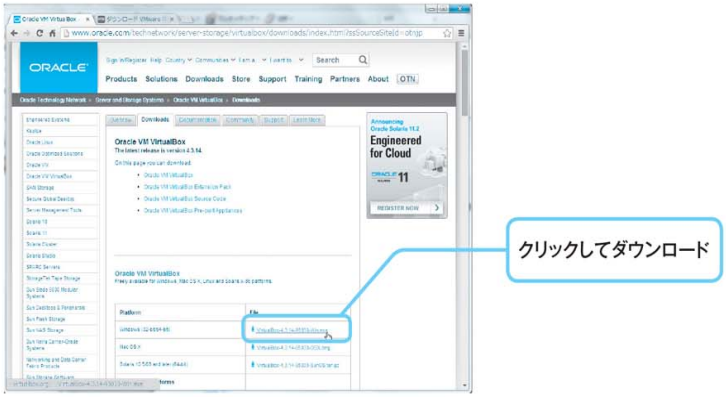
・ダウンロードしたVMイメージ（仮想OS）はVirtualBoxとVMware Player上で動作します（次ページ参照）．読者の方々のPC上で本書の内容を手軽に試すことができます．

●Oracle VM VirtualBox ／ VMware Playerの入手について

仮想OSを実行するツールを以下からダウンロードし，ダブルクリックしてインストールしてください．

「Oracle VM VirtualBox」（Windows/MacOSX/Linux）

http://www.oracle.com/technetwork/server-storage/virtualbox/downloads/index.html?ssSourceSiteId=otnjp



「VMware Player」（Windows/Linux）

https://my.vmware.com/jp/web/vmware/free#desktop_end_user_computing/vmware_player/6_0

入手しインストールしたツールに本書サポートサイトからダウンロードした「progtool-CentOS.ova」を読み込ませます．Oracle VM VirtualBoxでの詳しい手順については書籍22ページからの『1.3 VMを使ってみよう』で解説しています．

ここに記した内容は2014年7月末現在の内容です．サイトやサービスが変更してたどりつけない場合はウェブで「Oracle VM VirtualBox」や「VMware Player」で検索いただきダウンロードサイトを探して，ツールを入手してください．

Contents　目次

1

「体験」のための準備をしよう

[準備編 1]—VMとCentOS

まず最初に行うのは，プログラミングを行うためのツール群の準備です．つまり「開発環境」を，手持ちのＰＣに用意しましょう．

本書ではＣ言語プログラミングのために，「GCC」というコンパイラを利用します．そして開発環境には，「ＶＭ」というものを利用します．

まずは「コンパイラ」とは何か，そしてＶＭを利用したツール群の準備について説明します．

1.1 「コンパイラ」を知ろう

Ｃ言語によるプログラム開発を行うためには，「コンパイラ」というツールが必要になります．

まずは「プログラム」というものに対するイメージを最初に持っていただきたいので，とりあえず「Ｃ言語のプログラム」というものを見てみましょう．

例えば**リスト 1.1** は「Hello World!」という文字列を出力する，Ｃ言語のプログラムです．

ただし，ここで**リスト 1.1** の内容を理解する必要はありません．「こんな感じのものを書くのか」くらいの認識だけ，持っていただければ大丈夫です．

リスト 1.1: Ｃ言語のプログラムの例

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(void)
005: {
006:         printf("Hello World!\n");
007:         exit(0);
008: }
```

このように「Hello World」と表示するプログラムは一般に「ハローワールド」と呼ばれ，プログラミング言語に入門するときの定番です．「とりあえずハローワールドを書いてみた」などと言ったりします．

プログラムの詳細はまた後で説明しますが，ここではとりあえず，次のことだけを理解しておいてください．

- リスト1.1は「C言語」というプログラミング言語によって書かれた「プログラム」である.
- リスト1.1を実行すると,「Hello World!」と表示される.
- しかしリスト1.1をコンピュータ上でそのまま実行することはできない.
- 実行するにはまずリスト1.1の内容を「hello.c」のようなファイル名のファイルにして,さらにhello.cを「コンパイル」することで,「実行ファイル」を生成する.
- 生成した実行ファイルを実行すると,プログラムに書かれたとおりにコンピュータが動き出して,「Hello World!」と表示される.
- 「実行ファイル」に対してリスト1.1は,「ソースコード」と呼ばれる.

1.1.1 コンパイラとは何か?

コンピュータはC言語のようなプログラミング言語を直接理解して実行することはできません.つまりリスト1.1のプログラムを,いきなり実行することはできません.

コンピュータが理解し実行できるのは「機械語」「マシン語」などと呼ばれる言語のみです.

じゃあそういった「機械語」によってプログラムを書けばいいかというと,実はそういうわけにもいきません.機械語は単なる数字の羅列であり,コンピュータは理解できるのですが,我々人間にとっては非常に「とっつきにくい」言語になっています.不可能ではありませんが,たいへん不便です.

このため通常のプログラムはもっと人間にわかりやすい人間向けの言語で書き,それを機械語に変換(翻訳,などとも言われます)して実行ファイルとします.コンピュータは変換後の実行ファイルを実行することができます.このように変換を間に置くことで,人間によるプログラミングからコンピュータによる実行までに,段階を踏むようになっています.

この変換(翻訳)作業が「コンパイル」と呼ばれます.図1.1のような流れでプログラムを実行することになります.

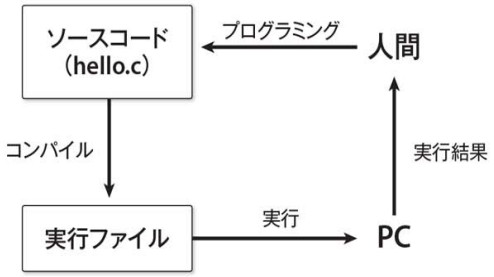


図1.1: プログラムの実行の流れ

コンパイルは「コンパイラ」というツールによって行われるため，我々人間が何か難しい作業をする必要は無く，「コンパイラを動かす必要がある」ということだけ知っていれば問題はありません．

1.1.2 フリーなコンパイラ「GCC」

プログラミング言語は様々な種類のものがあり，「コンパイラ」もそれらの言語ごとに必要です．

本書ではプログラミング言語として「C言語」を対象とします．よって「C言語コンパイラ」「Cコンパイラ」と呼ばれるツールをPCにインストールする必要があります．

本書では「C言語コンパイラ」に，GCCというコンパイラを利用します．

GCCは「GNU Compiler Collection」の略で，FSF（Free Software Foundation: フリーソフトウェア財団）という団体が推進している「GNUプロジェクト」によって開発されているコンパイラです．フリー（自由）に利用することができます．「ジーシーシー」と呼ばれます．

当初は「GNU C Compiler」の略でしたが，現在ではC言語以外のプログラミング言語も扱えるため，「GNU Compiler Collection」の略ということになっています．

1.1.3 GNU/Linuxディストリビューション

本書では簡単なプログラミングや，その他ツール類のお試し操作を体験してみます．

しかしプログラミングのためには，まずはコンパイラが必要です．さらにツール類の操作を体験するためには，もちろんそれらのツール類が必要になります．

こうしたプログラミング向けのツール群は「CentOS（セントオーエス）」や「Ubuntu（ウブントゥ）」といったいわゆる「GNU/Linuxディストリビューション」と呼ばれるシステム上で利用することができます．

もちろん「Windows」の上でも利用できます．また「Mac OS X」を愛用しているというかたもいるかもしれません．しかし「GNU/Linuxディストリビューション」はプログラマ向けの環境のためプログラミングに適しており，様々なプログラミング用ツールを親和性高く利用することができます．これはプログラミングを体験する上で，面倒が少ないということです．

本書では「CentOS」というシステムを利用します． CentOSはサーバなどでも利用されているため，操作を知っておくことは損ではないでしょう．そのようなシステムを使うのは初めてという読者のかたもいるかもしれませんが，良い機会ですので，ここで慣れてしまいましょう．

1.2 VM（Virtual Machine）を知ろう

CentOSやUbuntuといったシステムはコンピュータを利用するためのひとつの環境ですので，WindowsのかわりにＰＣにインストールして利用することができます．

ここで注意しなければならないのは，「Windowsの上にインストールする」ということではなく，「Windowsのかわりにインストールする」ということです．ＰＣの電源を入れたらWindowsが起動するのでなく，CentOSという完全に別のシステムが起動するというイメージです．

しかしそのためには，ＰＣ上からWindowsを削除する必要が出てしまいます．実は「デュアルブート」といって，うまくハードディスクを分割してWindowsと共存させた状態でインストールすることもできたりはします．これだとＰＣの電源を入れたときに，WindowsとCentOSのどちらを起動するかを選択する感じになります．

ただ，プログラミングを試すだけのためにＰＣをそこまで色々いじるというのはなかなか難しいでしょう．またインストール作業も，（インストール自体はそれほど難しくはないのですが）その後の設定やツール類の追加インストールまでも含めると，プログラミングの前段階として気軽にできるというわけでもありません．

そこで本書では「ＶＭ」と呼ばれるシステムを利用します．

1.2.1 「VM」とは何か?

「ＶＭ」は「Virtual Machine」の略で，仮想ＰＣのことです．つまりＰＣの上で仮想的に動作する，ソフトウェア的なＰＣです．たとえばWindowsの上でＶＭが動作してウィンドウがひとつ開き，そのウィンドウの中でもうひとつのＰＣが仮想的に動作している，というイメージです．

仮想ＰＣといえどＰＣですので，そこには何らかのＯＳをインストールして動かすことができます．つまり「CentOS」や「Ubuntu」をインストールして動作させることができるわけです．

そしてＶＭの便利なところは，ＯＳとしてCentOSなどをインストール済みのハードディスクのイメージを，ひとつの巨大なファイルとして扱うことができるという点です．

そのファイルを別のＰＣ上にコピーして，新たに起動することもできます．つまりＶＭ上に「CentOS」や「Ubuntu」などをインストールし，さらに様々なツール類も追加インストール済みの状態で，そのハードディスクのイメージをファイルとしてまるごと受渡しすることができるという

ことです.(もちろん配布の際には,インストール物のライセンスには別途留意する必要があります)

このようにして筆者が作成したCentOSのＶＭイメージのファイルをインターネット上で配布しています．本書の内容をそのまま試せるように，必要なツール類もすべてインストール済みです．このＶＭイメージをダウンロードし，ＶＭ上で動作させることで，読者の方々のＰＣ上で本書の内容を手軽に試すことができます．

1.2.2 VMをインストールする

ＶＭには無料で配布されているものや製品版のものなど，いくつかがあります．

一般には「VirtualBox（バーチャルボックス）」と「VMware Player（ヴィエムウェア プレイヤー）」の２つに人気があるようです．ここではこれら２つを紹介しておきましょう．

VirtualBoxは現在はオラクルによって開発され，オープンソースで配布されています．

```
https://www.virtualbox.org/
```

VMware Player は VMware, Inc により開発され, 非営利目的のユーザは無償で利用できるというものです．

```
https://www.vmware.com/jp/
```

読者の方々が本書用のＶＭイメージを利用するためには，ＰＣ上になんらかのＶＭをインストールする必要があります．イメージ自体は VirtualBox と VMware Player の両方で動作を確認済みですので，これら２つのいずれかをインストールするのがいいでしょう．

インストールするのはどちらでも構いませんが，インターネット上に様々なレビューなどが出ていますので，気に入ったほうをインストールしてください．なお利用の際には，それぞれの利用許諾条件などに留意してください．

本書では，VirtualBox を利用して説明を進めます．

これらのインストール方法については，ここでは詳しく説明しません．インストール自体はさほど難しいものではありませんし，インターネット上で検索すれば様々な情報も得られます．ぜひ，チャレンジしてみてください．

1.3 VMを使ってみよう

VMのインストールは無事に完了したでしょうか.

VMがインストールできたら, VMイメージを利用するための土台はできました. 次はVMイメージの利用です.

これはVMイメージをダウンロードし, さらにVMに「インポート」することで可能です.

1.3.1 VMイメージをダウンロードする

まずはVMイメージをダウンロードします.

VMイメージは以下の2つサイトで配布されています.「progtool-CentOS.ova」というファイルをダウンロードしてください(どちらからダウンロードしても内容は同じです). ファイルサイズは1.2GBほどありますので, ダウンロード時間やハードディスク容量などに注意してください.

```
http://kozos.jp/vmimage/progtool.html
http://book.mynavi.jp/support/pc/5149/
```

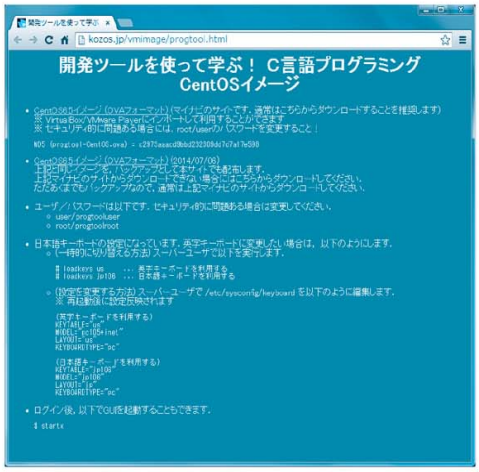


図1.2: http://kozos.jp/vmimage/progtool.html

「progtool-CentOS.ova」はOVA形式と呼ばれるファイルになっており, VirtualBox もしくは VMware Player にインポートして利用することができます.

1.3.2 VirtualBoxにVMイメージをインポートする

VMイメージをダウンロードできたら，VMにインポートしてみましょう．ここではVirtualBoxでの方法を説明します．

VirtualBox を起動すると，**図 1.3** のようなウィンドウが開きます．

図 1.3: VirtualBox の起動画面

ここで「ファイル」メニューから「仮想アプライアンスのインポート」を選択します．
すると**図 1.4** のようなウィンドウが開きます．

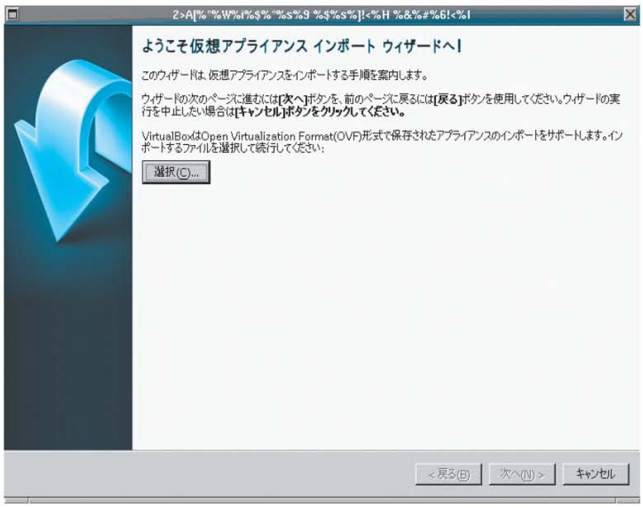


図 1.4: 仮想アプライアンスのインポートを選択する

インポートするファイルとして先ほどダウンロードしたOVAファイルである「progtool-CentOS.ova」を選択し，「次へ」をクリックします．すると**図1.5**のようにして，OVAファイルの情報の確認画面になります．

図1.5: OVAファイルの情報を確認する

このままでとくに問題は無いはずなので，そのまま「完了」をクリックします．するとインポートが開始されます．

図1.6: インポートには数分かかる

インポートには多少の時間がかかります．**図1.6**のようにプログレスバーが表示されますので，完了までしばらく待ちます．

インポートが完了すると**図1.7**のような画面になり，左上に「progtool-CentOS」のアイコンが出現します．最初は「電源オフ」の状態になっているはずです．

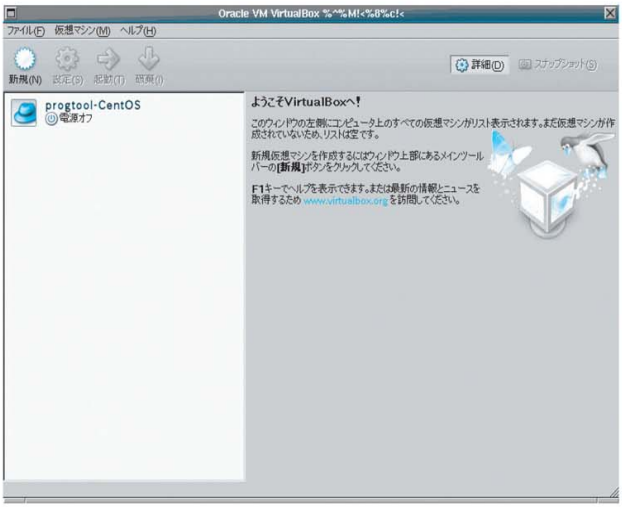


図1.7: インポートが完了したところ

左上には「新規」の隣に「設定」「起動」などのメニューもありますが，そのままではハイライトされておらず，選択できません．ここで「progtool-CentOS」のアイコンをクリックしてみましょう．

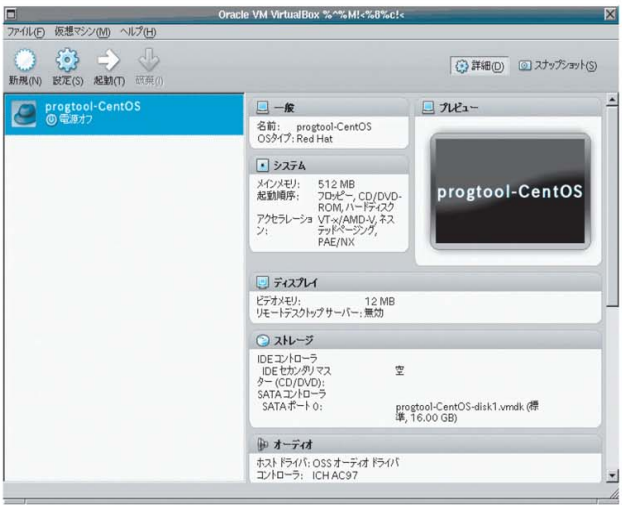


図1.8: VMをクリックして選択する

すると**図1.8**のようにして，「設定」や「起動」がハイライトされ，選択できるようになります．

「設定」をクリックすると**図1.9**のようなウィンドウが開き，各種設定が行えます．本書ではとりあえず設定変更の必要は無いため，これはこのままキャンセルしてしまって構いません．

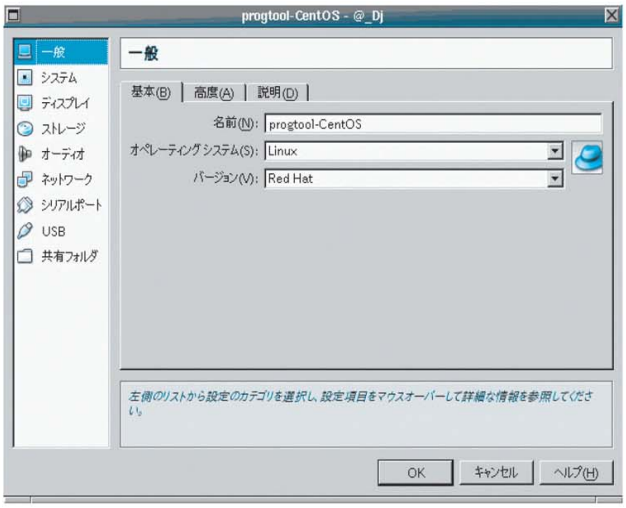


図1.9: 設定の確認

1.3.3 VMware Player の場合

VirtualBoxでのインポート方法について説明しましたが，VMware Player での方法についても簡単に触れておきましょう．

VMware Player の場合は「仮想マシンを開く」のメニューからOVAファイルを選択することで，インポートできます．

インポートの際には「チェックに合格しなかったため，インポートに失敗しました」のように言われることがあります．この場合は気にせずに「再試行」をクリックすることで，チェックを緩和してインポートが再試行されます．

インポート後には「progtool-CentOS」のアイコンが出現しますので，「仮想マシンの再生」をクリックすることでインポートしたCentOSを起動できます．

1.3.4 CentOSを起動してみる

ＶＭイメージがインポートできたところで，その中にインストールされている「CentOS」を，いよいよ起動してみましょう．

図1.8の画面で「起動」をクリックすると，**図1.10**のように１枚のウィンドウが開きます．画面

上部に「Press any key to enter the menu」のように表示され，「in 1seconds...」のようにカウントが始まります．しかしここで何かキーを押してしまうと起動メニューに入ってしまうので，何もキーを押さずにそのままやりすごします．

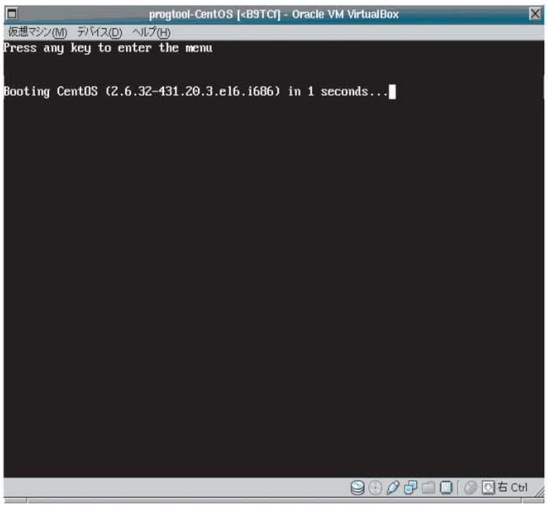


図 1.10: 起動すると，まずカウント画面が表示される

もしも何かキーを押してしまった場合には，**図 1.11** のようなメニュー画面に入ります．この場合には，そのまま Enter を押してください．

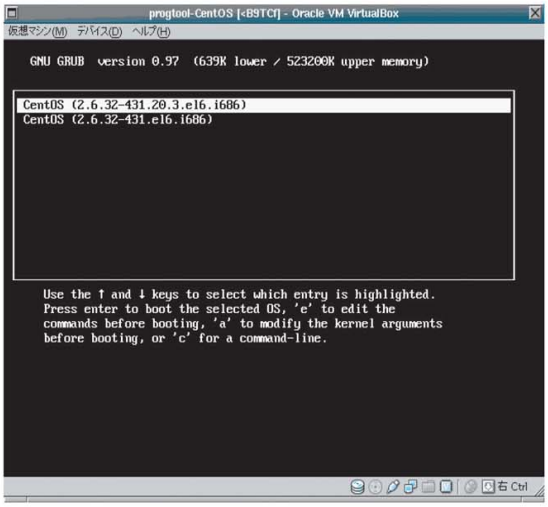


図 1.11: 起動時のメニュー画面

すると**図1.12**のような画面になり，VMイメージにインストール済みのCentOSが起動を始めます．

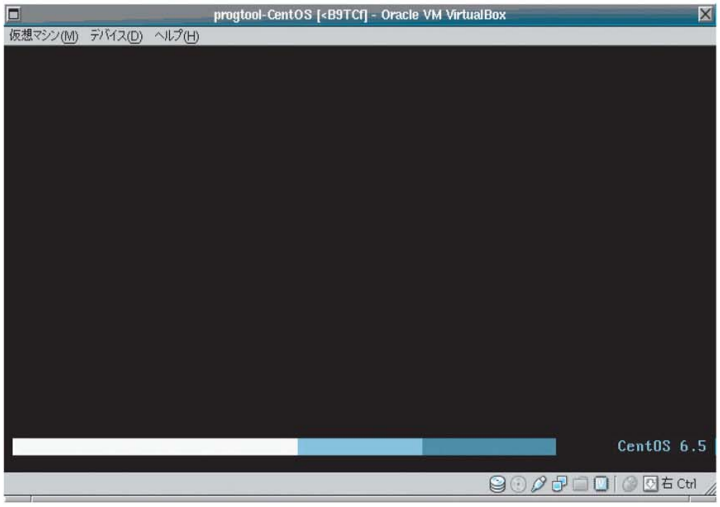


図1.12: CentOSの起動画面

しばらくすると，**図1.13**のようになります．これは起動が完了し，ユーザのログイン待ちになっているところです．

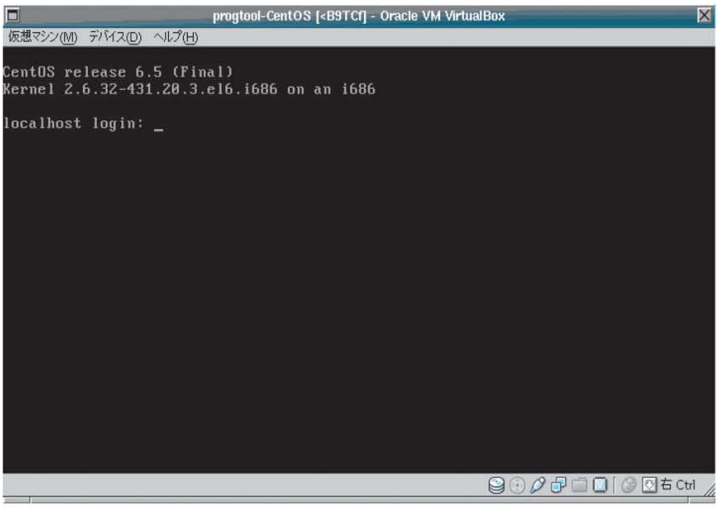


図1.13: 起動が完了すると，ログイン待ちの画面になる

Column

ＶＭの画面内のCentOSに対してキーボード入力を行うには，ＶＭのウィンドウにマウスカーソルを合わせて左クリックします．するとＶＭのウィンドウがハイライトされ，キーボードの入力がCentOS側に伝えられるようになります．

このとき，マウスの入力もCentOS側に伝えられるようになり，ウィンドウ外の通常の操作ができなくなってしまいます．これを解除するには「ホストキー」を押します．

図1.13の右下には，「右Ctrl」の文字があります．これが「ホストキー」と呼ばれるものです．**図1.13**では「右Ctrl」がホストキーとなっているので，キーボード右側の「Ctrlキー」を押すことでウィンドウ内から外に抜け出ることができます．Ctrlキーは左右に２つありますが，右と左は区別されます．

ホストキーがわからないとＶＭ内から抜けられなくなるので，ホストキーがどのキーに割り当てられているのかは，ＶＭの操作前に確認しておくといいでしょう．通常はキーボード右側のCtrlキーかAltキーなどに割り当ててあることが多いようです．またホストキーは設定で変更することが可能ですので，使いやすいキーに割り当てておくといいでしょう．

1.4 CentOSにログインしよう

CentOSが起動すると，画面に以下のように表示されているはずです．

```
localhost login:
```

これはユーザのログイン待ち状態になっているということです．

ログインしてみましょう．

1.4.1 ログインしてみよう

本書で利用するCentOSのＶＭイメージには，以下のユーザが登録されています．

- **ユーザ名:「user」**
- **パスワード:「progtooluser」**

まずは**図1.14**のように，キーボードからユーザ名である「user」を入力します．打ち間違えをしたときには，「Backspace」のキーを押すと１字戻すことができます．「Backspace」のキーはキーボードの右上にあります．

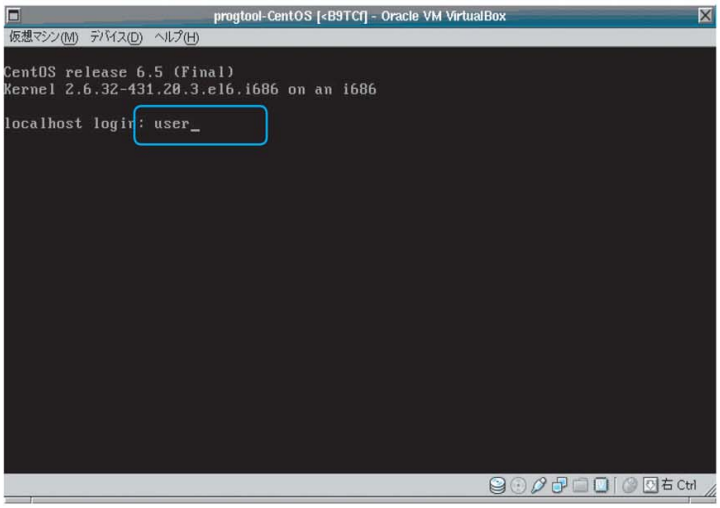


図1.14:「user」と入力する

入力したら「Enter」のキーを押します．「Enter」はキーボードの右側にある，大きめのキーです．「←┘」のような表記がされていたりもします．

「Enter」を押すことで入力したユーザ名が確定し，**図 1.15** のようなパスワード入力画面になります．

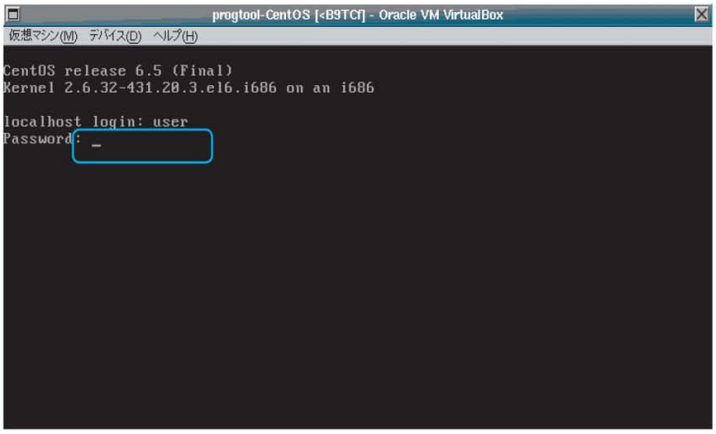


図 1.15: パスワードの入力画面

ここでパスワードとして「progtooluser」を入力します．パスワード入力ですので，入力した文字は実際には画面に表示されません．なので打った文字が画面に出なくても気にせずに入力を続け，Enter キーを押します．

Enter を押して**図 1.16** のような画面になれば，ログイン完了です．

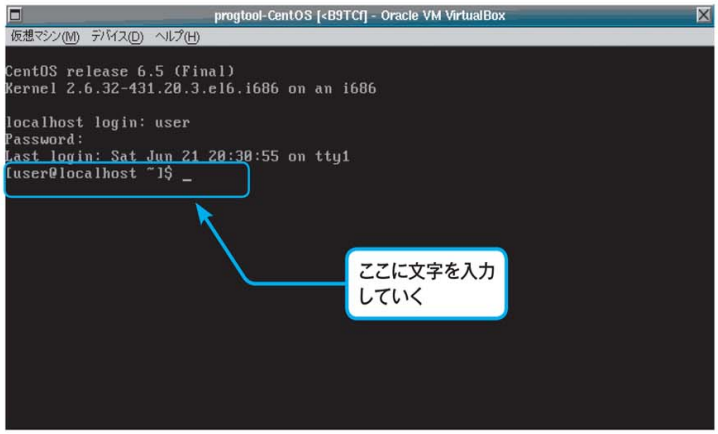


図 1.16: ログイン完了！

なおユーザ名やパスワードを打ち間違えた場合には，**図1.17**のように「Login incorrect」と表示されて再度ユーザ名の入力待ちになります．この場合にはユーザ名の入力からもう一度やりなおしてください．

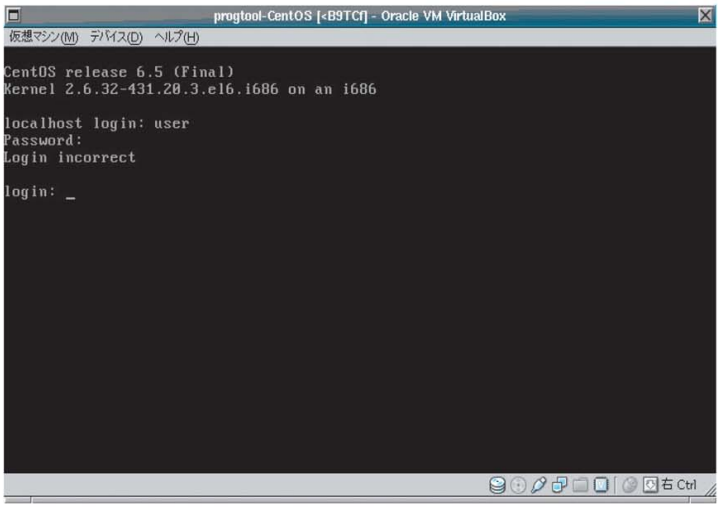


図1.17: ユーザ名やパスワードを間違った場合には，ログイン失敗する

1.4.2 コマンド操作を体験しよう

ログインが成功すると，**図1.16**のような画面になります．見ると，以下のような1行が出てきています．

```
[user@localhost ~]$
```

これは「コマンドプロンプト」または単に「プロンプト」などと呼ばれるもので，コマンド入力を待っている，ということを示しています．

図1.16を見ると，実際には「$」の右側にはアンダーバーが出ています．これは「カーソル」と呼ばれるものです．「カーソル」は，入力した文字がそこに書き込まれるという目印です．

ではここで「whoami」と入力してみましょう．

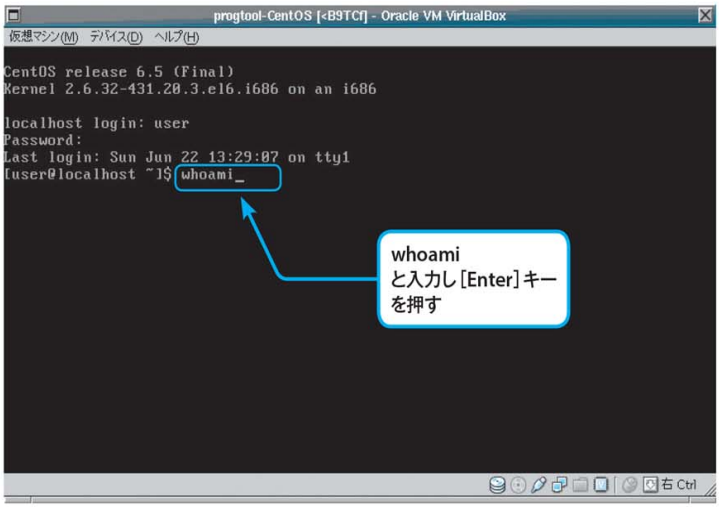


図1.18:「whoami」と入力する

キーボードの「W」「H」「O」…というキーを打鍵していくことで，入力した文字がカーソルの位置に表示され，カーソルはひとつ右に移動します．打ち間違えは，やはり「Backspace」で戻すことができます．

このように，入力した文字はカーソルの位置に書き込まれていきます．「whoami」と入力すると一文字の入力ごとにカーソルが右に移動し，最終的には**図1.18**のようになります．

「whoami」と入力したら，続けてEnterキーを押してみましょう．すると**図1.19**のように，「user」と表示されるはずです．

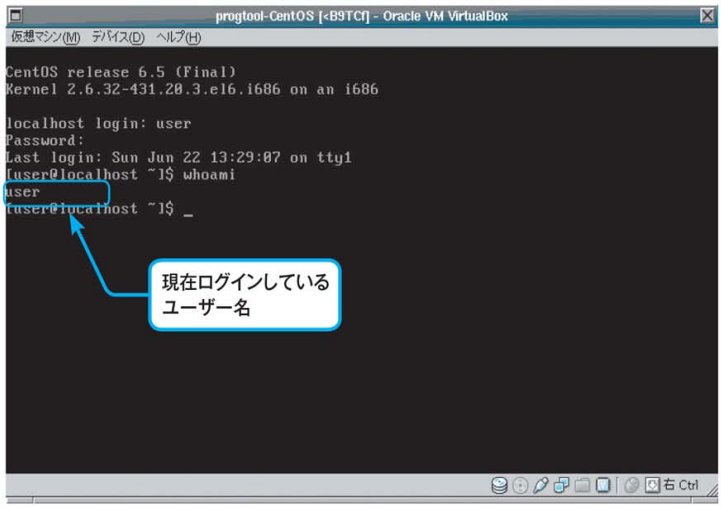


図1.19:「whoami」を実行する

これで「whoami」というコマンドを実行したことになります．

whoamiは現在のユーザ名を表示するコマンドです．今は「user」というユーザ名でログインしていますので，結果としてユーザ名の「user」が出力されています．

このようにコマンドは，実行したいコマンドを入力し，Enterを押して確定するという手順で実行します．このような入力行は「コマンドライン」，入力方法は「コマンドライン入力」などと呼ばれます．

Column

本書のＶＭイメージのCentOSは，日本語キーボードを利用するものとして設定されています．このため英字キーボードのＰＣを利用している場合には，「@」や「＋」などの記号文字がうまく入力できません．

p.39で説明する「スーパーユーザ」になり，以下を実行することで一時的に英字キーボードの利用に切り替えられます．次節ではパスワードの変更を行いますが，キーボードの切り替えを行わずにパスワードを変更した場合，パスワードに記号文字を使うとキーボードの切り替え後に入力できなくなってしまいますので，注意してください．

（スーパーユーザで以下を実行することで，キーボードを切り替える）

```
[root@localhost user]# loadkeys us      ... 英字キーボードを利用する
[root@localhost user]# loadkeys jp106   ... 日本語キーボードを利用する
```

ただし上の切り替えは，CentOSを再起動すると元に戻ってしまいます．再起動後も切り替えを有効にするには，スーパーユーザで以下を実行してテキストエディタ（4章で説明します）を起動し，キーボードの設定ファイルを以下のように編集・保存してください．

CentOSを再起動後することで，設定反映されます．（テキストエディタの使いかたがよくわからなければ，とりあえず上のコマンドで一時的に変更することで対応して，4章を読んでから改めて設定を行ってください）

（スーパーユーザでテキストエディタを起動し，キーボードの設定ファイルを編集する）

```
[root@localhost user]# nano /etc/sysconfig/keyboard
```

（英字キーボードを利用する設定）

```
KEYTABLE="us"
MODEL="pc105+inet"
LAYOUT="us"
KEYBOARDTYPE="pc"
```

（日本語キーボードを利用する設定）

```
KEYTABLE="jp106"
MODEL="jp106"
LAYOUT="jp"
KEYBOARDTYPE="pc"
```

1.4.3 ログアウトの方法

ログインできたら，ひとまずログアウトの方法も覚えておきましょう．

ログアウトは「exit」というコマンドで可能です．以下のように「exit」とコマンドラインに入力し，Enterを押してみます．

```
[user@localhost ~]$ exit
```

exitを実行すると，再び**図1.20**のようにログイン待ちの状態に戻ります．

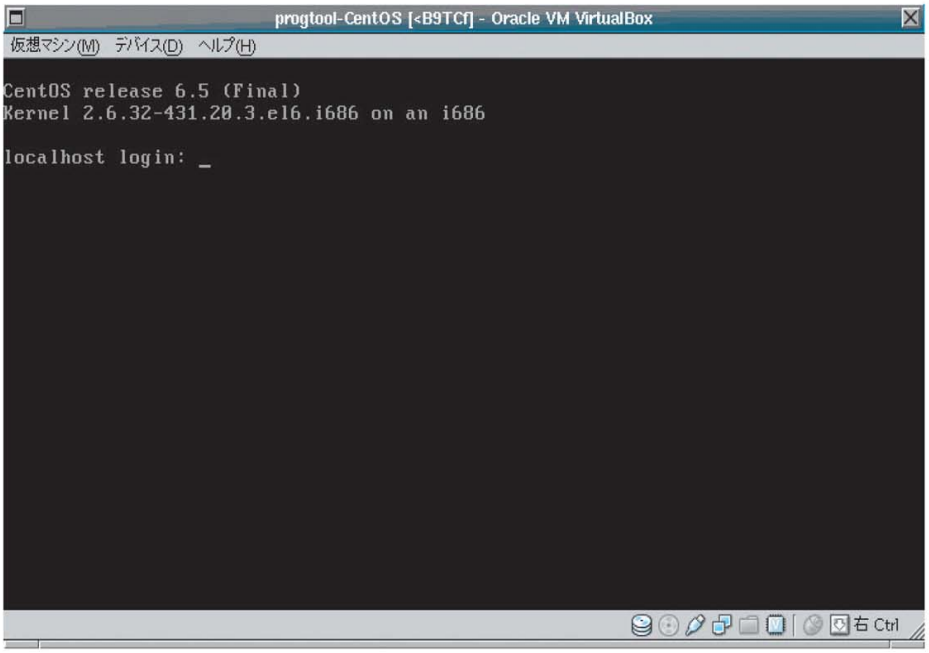


図1.20: exitすると，再度ログイン待ちの画面になる

1.5 パスワードを変更してみよう

このままではユーザ名とパスワードは書籍用の共通のものになっていますので，そのまま使い続けるにはセキュリティ的な不安があります．

ログインができるようになってまず行うことは，パスワードの変更です．

1.5.1 パスワードを変更しておこう

パスワードは「passwdコマンド」で変更できます．まずは「user」でもう一度ログインしなおしましょう．パスワードには「progtooluser」を入力します．

```
localhost login: user
Password:
```

ログインしたら，以下のように「passwd」と入力してEnterを押してみてください．これでパスワード変更のコマンドである「passwdコマンド」を実行することができます．

```
[user@localhost ~]$ passwd
Changing password for user user.
Changing password for user.
(current) UNIX password:
```

passwdコマンドが実行されると，パスワード変更の前に確認として「(current) UNIX password」のように，現在のパスワードを聞かれます．ここでは現在のパスワードとして「progtooluser」を入力します．

すると「New password」として，新しいパスワードを聞かれます．

```
[user@localhost ~]$ passwd
Changing password for user user.
Changing password for user.
(current) UNIX password:
New password:
```

ここで自分の好きなパスワードを入力して，Enterを押してください．なお入力したパスワードは画面には表示されませんので，画面には出なくても気にせずに全部入力します．

```
[user@localhost ~]$ passwd
Changing password for user user.
Changing password for user.
(current) UNIX password:
New password:
Retype new password:
```

うまく入力できると，「Retype new password」のように確認のための再入力を求めてきます．ここで同じパスワードをもう一度入力します．

すると以下のようになり，パスワードの変更は完了します．

```
[user@localhost ~]$ passwd
Changing password for user user.
Changing password for user.
(current) UNIX password:
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
[user@localhost ~]$
```

変更したパスワードは今後のログインで必要になりますので，忘れないように覚えておきましょう．

なおパスワードが単純すぎたり短すぎたりする場合には，以下のように「BAD PASSWORD」と表示されて再入力を促してきます．

```
New password:
BAD PASSWORD: it is WAY too short
New password:
```

また確認のために再入力したパスワードが一致しない場合には，以下のようにエラーとなり，やはり再入力を促してきます．

```
New password:
Retype new password:
Sorry, passwords do not match.
Password:
```

1.5.2 変更したパスワードを確認しよう

さて，パスワードを変更したら，期待通りに変更できたかどうかを確認しておきたいところです．

いったんログアウトしてからログインしなおせば，パスワードが変更されたかどうかを確認することができます．しかしこれは危険です．なぜならば，もしも間違ったパスワードを入力してしまっていた場合，二度とログインできないことになってしまうかもしれないからです．

このためここではログアウトをせずに，passwdコマンドをもう一度実行することで確認しましょう．

```
[user@localhost ~]$ passwd
Changing password for user user.
Changing password for user.
(current) UNIX password:
```

現在のパスワードを聞いてきますので，まずは変更前のパスワードである「progtooluser」を入力してみます．

```
[user@localhost ~]$ passwd
Changing password for user user.
Changing password for user.
(current) UNIX password:
passwd: Authentication token manipulation error
[user@localhost ~]$
```

エラーになり終了しています．これはパスワードが標準の「progtooluser」から，先ほど設定したものにすでに変更されているためです．

次にもう一度 passwd コマンドを実行し，今度は変更後のパスワードを入力してみます．

```
[user@localhost ~]$ passwd
Changing password for user user.
Changing password for user.
(current) UNIX password:
New password:
```

今度はパスワードが正しかったためチェックを通過し，新しいパスワードの入力待ちになっています．パスワードは期待通りに変更されているようです．

「New password」と聞かれていますが，このまま何か入力すると，またパスワードが変更されてしまいます．このためpasswdコマンドを「Ctrl」＋「C」で強制終了します．

「Ctrl」＋「C」は，「Ctrl キー」を押しながら「C」のキーを押す，という意味です．Ctrl キーはキーボードの左下にあります．実際に押すと以下のようになり，passwd コマンドを強制終了できます．「Ctrl」＋「C」はプログラムの強制終了によく使われる操作ですので，覚えておくといいでしょう．

```
[user@localhost ~]$ passwd
Changing password for user user.
Changing password for user.
(current) UNIX password:
New password:
[user@localhost ~]$
```

さて，もしも変更後のパスワードを入力してもエラーとなってしまった場合には，パスワード変更時に入力間違えをしてしまった可能性があります．この場合，心当たりのある「間違ったパスワード」をいくつか試してみてください．

それでも駄目な場合は，ログアウトする前に次で説明する「スーパーユーザ」になってから「passwd user」を実行することで，「user」のパスワードを変更することができます．（もしくは面倒で無ければ，ＶＭを削除してインポートしなおすことでやり直すことができます）

1.6 スーパーユーザになってみよう

これで「user」というユーザのパスワードは変更できたのですが，実は「user」以外にも，もうひとつ重要なユーザがあります．

それは「スーパーユーザ」というものです．「スーパーユーザ」はシステムの管理用の，特権ユーザです．いわゆる「管理者権限」を持っているユーザで，「adminユーザ」などとも呼ばれるものです．

これに対して，先ほどパスワードを変更した「user」というユーザは「一般ユーザ」と呼ばれるものです．プログラミングなどの普段の作業は一般ユーザで行い，システムの設定変更などはスーパーユーザで行います．

1.6.1 スーパーユーザに切り替える

ということで，一般ユーザだけではなくスーパーユーザのパスワードも変更しておきましょう．

しかしそのためにはユーザを「一般ユーザ」から「スーパーユーザ」に切り替える必要があります．まずはその方法を説明しましょう．

スーパーユーザになるには，以下のように「su」というコマンドを実行します．

```
[user@localhost ~]$ su
Password:
```

ここでスーパーユーザのパスワードを聞いてきます．本書で利用するCentOSのVMイメージでは，スーパーユーザのパスワードは以下のようになっていますので，そのように入力します．

- **ユーザ名:「root」**
- **パスワード:「progtoolroot」**

これもやはり入力しても画面には表示されませんので，入力文字が画面に出なくても気にせずに入力してEnterを押します．

パスワードが入力できると，次のようにプロンプトが切り替わります．

```
[user@localhost ~]$ su
Password:
[root@localhost user]#
```

プロンプトの右端が「$」から「#」に変化していることに注目してください．これでスーパーユーザになっています．

whoamiで，ユーザ名を確認してみましょう．

```
[root@localhost user]# whoami
root
[root@localhost user]#
```

先ほどは「user」と表示されていましたが，今度は「root」と表示されました．この「root」が，スーパーユーザのユーザ名です．このためスーパーユーザになることを「rootになる」と言ったり，スーパーユーザの権限のことを「root権限」と言ったりもします．

スーパーユーザから一般ユーザに戻るには，exitコマンドを実行します．一度，戻ってみましょう．

```
[root@localhost user]# exit
exit
[user@localhost ~]$
```

もう一度，whoamiでユーザ名を確認してみます．

```
[user@localhost ~]$ whoami
user
[user@localhost ~]$
```

ユーザ名が「user」に戻っています．つまりいったんスーパーユーザになったが，exitすることで一般ユーザに戻っているわけです．

1.6.2 スーパーユーザのパスワードも変更しよう

スーパーユーザに切り替えることができたところで，スーパーユーザのパスワードも変更しておきましょう．

手順は先ほどと同じで，スーパーユーザになってpasswdコマンドを実行するだけです．

まずはもう一度，suコマンドでスーパーユーザになります．パスワードは「progtoolroot」です．

```
[user@localhost ~]$ su
Password:
[root@localhost user]#
```

スーパーユーザになったら，passwdコマンドを実行します．

```
[root@localhost user]# passwd
Changing password for user root.
New password:
```

これも，自分の好きなパスワードを入力してください．

```
[root@localhost user]# passwd
Changing password for user root.
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
[root@localhost user]#
```

これでスーパーユーザのパスワード変更は完了です．これは「管理者パスワード」に相当するものになりますので，忘れずに覚えておきましょう．

1.6.3 スーパーユーザのパスワードを確認しよう

スーパーユーザのパスワードを設定したら，これも確認をしておきましょう．

しかしスーパーユーザでpasswdコマンドを実行しても，現在のパスワードの入力無しで変更ができてしまうため，確認ができません．ということでここでは，以下の手順で確認してみます．

- **スーパーユーザから一般ユーザに一時的に切り替える**
- **切り替えた一般ユーザから再度スーパーユーザになってみる**

まず以下のように「su user」を実行して，一般ユーザに切り替えます．

```
[root@localhost user]# su user
[user@localhost ~]$
```

suコマンドはこのようにユーザ名を指定して実行すると，そのユーザに切り替えるという意味になります．これは「コマンドライン引数」と呼ばれるものです．コマンドライン引数については後でまた説明しますので，ここではそのようなものとして理解しておいてください．

ユーザが切り替わっていることを，whoamiで確認しましょう．

```
[user@localhost ~]$ whoami
user
[user@localhost ~]$
```

問題なく切り替わっているようです．

ここで「su」を実行します．

```
[user@localhost ~]$ su
Password:
```

今は「user」の一般ユーザになっていますので，スーパーユーザになろうとしてパスワードが聞かれています．ここで，変更前のスーパーユーザのパスワードである「progtoolroot」を入力してみます．

```
[user@localhost ~]$ su
Password:
su: incorrect password
[user@localhost ~]$
```

「incorrect password」と出力されていますので，パスワードが不正となっています．

次にもう一度suコマンドを実行し，今度は変更後のパスワードを入力してみます．

```
[user@localhost ~]$ su
Password:
[root@localhost user]#
```

今度はスーパーユーザになることができました．

さて，現在は一般ユーザでログインし，スーパーユーザになり，「su user」で一般ユーザとなり，さらにまたスーパーユーザになったという状態です．このためもしもパスワード設定を誤ってしまって，スーパーユーザになれなかった場合には，exitで一般ユーザから抜けることでスーパーユーザに戻り，再度passwdコマンドでパスワードを設定しなおすことができます．

パスワードが正常に設定できていることが確認できた場合には，「exit」を実行することで，多重にユーザ切り替えした状態からひとつずつ抜けていくことができます．

まず「exit」を一回実行してみましょう．

```
[root@localhost user]# exit
exit
[user@localhost ~]$
```

プロンプトが「$」になっていますので，一般ユーザに戻っているようです．続けてもう一度，exitを実行します．

```
[user@localhost ~]$ exit
exit
[root@localhost user]#
```

これは「su user」を実行する前の，スーパーユーザに戻った状態です．

1.6.4 なぜ「スーパーユーザ」があるのか?

さて，ここまででユーザには「user」と「root」の2つがあることを説明しました．

- **「user」は一般ユーザと呼ばれるもので，他にも「user2」などを定義することもできます．**
- **そして「root」はスーパーユーザと呼ばれるもので，システムの変更はスーパーユーザで行います．これは一見して，面倒なように思えるかもしれません．**

しかし普段のプログラム開発などは一般ユーザで行い，システムの変更などはスーパーユーザで行うことで，うっかりしたミスでシステムに不要な影響を与えてしまうことをある程度防ぐことができます．

「普段の開発作業は一般ユーザ」「システムの変更はスーパーユーザ」のように，使い分けるようにしましょう．

Column

ＶＭを起動しているＰＣがネットワークに接続されている場合には，本書のＶＭ上のCentOSもネットワークを利用できます．ただしネットワークを利用する場合には，セキュリティを考慮して，CentOSを最新版に更新しておいたほうがいいでしょう．

ネットワークに接続された状態で，スーパーユーザで以下を実行することで，CentOSのシステムを最新に更新することができます．

```
[root@localhost user]# yum -y update
```

なおインターネット接続にプロキシを経由している場合には，上記コマンドの実行前に，スーパーユーザで以下を実行して設定ファイルを編集・保存することで，プロキシ経由で更新が行えるようになります．（テキストエディタの使いかたの説明は4章にあります）

（スーパーユーザでテキストエディタを起動し，設定ファイルを編集する）

```
[root@localhost user]# nano /etc/yum.conf
```

（yum.confの末尾に，以下の行を追加する）

```
proxy=http://(プロキシのURL):(ポート番号)/
proxy_username=(プロキシのユーザ名:必要な場合のみ)
proxy_password=(プロキシのパスワード:必要な場合のみ)
```

1.6.5 シャットダウンで終了しよう

ここまで確認できれば，作業はひとまずひと段落です．

しかしCentOSは，いきなり終了させてはいけません．終了時には，必ずシャットダウン処理を行うようにしましょう．

スーパーユーザで「init 0」を実行することで，シャットダウンできます． suコマンドでスーパーユーザになり，init 0 を実行してみましょう．

```
[user@localhost ~]$ su
Password:
[root@localhost user]# init 0
```

これで図1.21のような画面になり，シャットダウン処理が始まります．シャットダウンが完了すると，VMのウィンドウが自動的にクローズします．

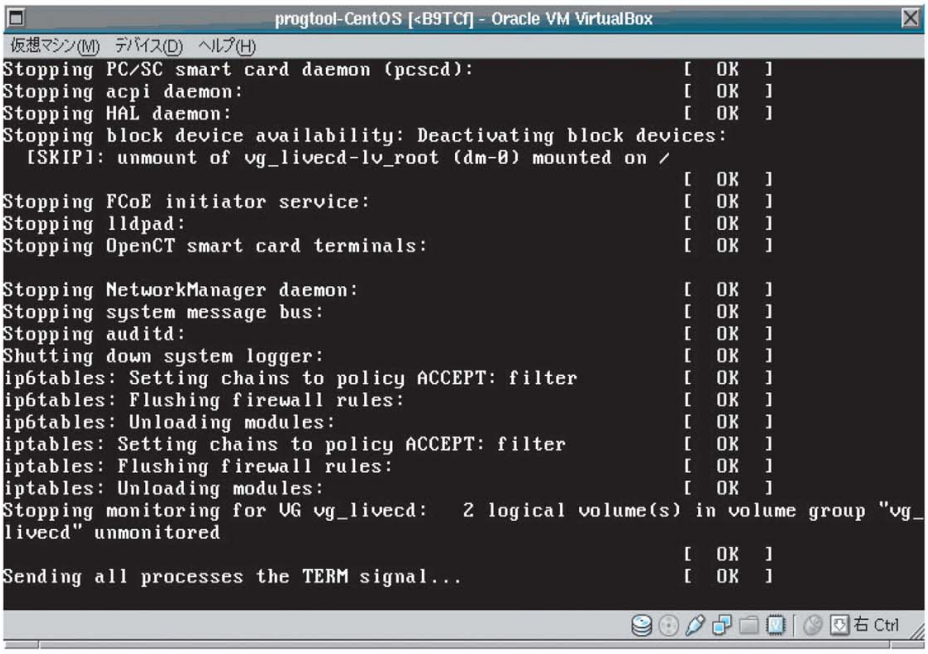


図1.21: シャットダウン処理中の画面

Column

本書の内容は，本章で説明したような「コマンドでの操作」をベースにしています．つまりGUIによるグラフィカルな画面での操作は行いません．

しかし実は「GUIを使いつつ，その中でコマンドを使う」という方法もあります．その方法を説明しておきましょう．なお実際に本書を読み進める際には，どちらのやりかたでも構いません．

GUIを利用するには，ログイン後に以下のコマンドを実行します．

```
[user@localhost ~]$ startx
```

すると図 1.22 のような画面が起動します．

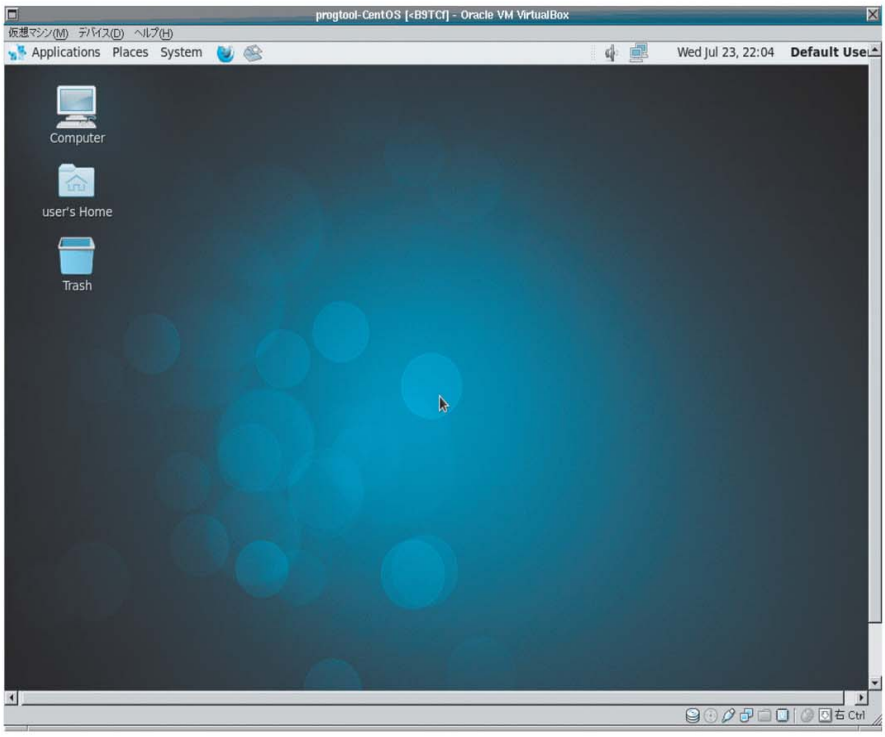


図 1.22: GUIを起動する

コマンドを利用するには，左上の「Applications」のメニューをクリックし，さらに「System Tools」→「Terminal」のように選択することで，**図1.23**のような「ターミナル」が開きます．

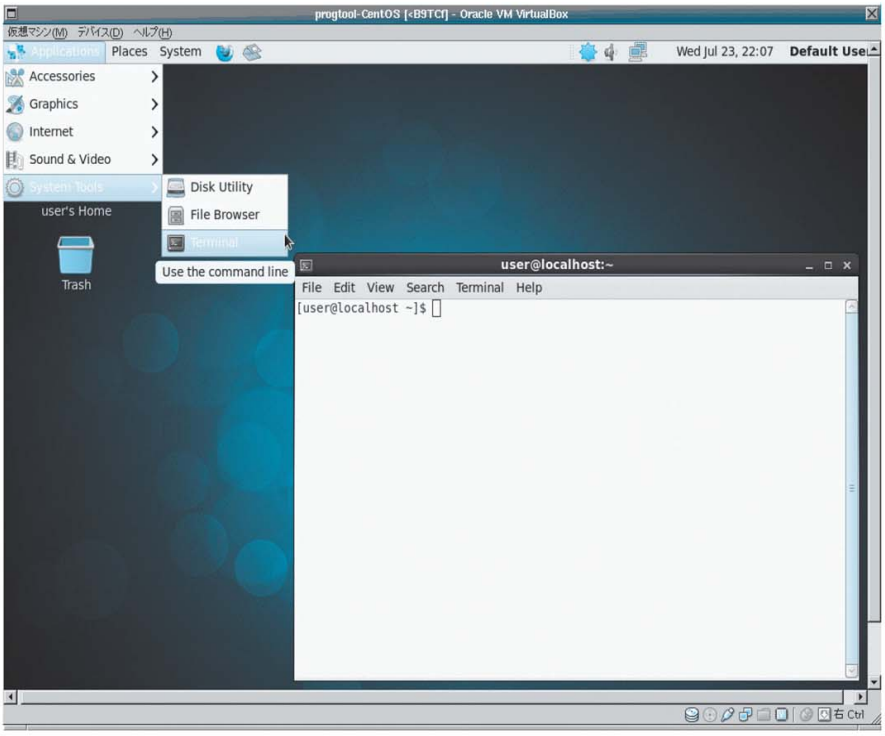


図1.23: ターミナルを起動する

ターミナルのウィンドウ内には，コマンドプロンプトが表示されています．この「ターミナル」の中でも本章で説明したようなコマンドが実行できますので，本書の内容を試すことができます．好きなやりかたで読み進めていってください．

なお画面左上の「System」のメニューから，システムの設定もGUI上で行えます．キーボードの設定変更なども，ここで可能です．

またGUIを起動した場合にも，p.28で説明した「ホストキー」を押すことで，VMのウィンドウ内から外に出ることができます．

1.7 まとめ

最初の章ではＶＭ上でCentOSを起動してログインし，ユーザ名の確認やパスワードの変更などの簡単なコマンド操作を体験してみました．

コマンド実行は，コマンド文字列をキーボードから入力してEnterキーで確定するという，コマンドライン入力によって行いました．このようなキーボードからのコマンド入力を主体とした操作は慣れない読者のかたも多いかもしれません．

しかしこれは，サーバ操作などに応用できる方法でもあります．今どきはサーバの操作もGUIやブラウザ経由でできたりもするのですが，やはりいざというときに役に立つのは「コマンドライン入力」です．

ぜひこの機会に「コマンドライン」に慣れてみてください．

2

シェルの操作を覚えよう

[準備編2]—コマンド操作

開発環境が用意できたら，まずは基本的なファイル操作の方法を覚えましょう．

コンピュータを扱うときに，その操作のほとんどは実はファイルに関するものです．ファイルの作成やコピー・移動・削除などです．

これは，結局のところ我々がコンピュータで扱いたいのは「データ」であり，データはコンピュータ上では「ファイル」として名前をつけて管理されているためです．

GUIでは，ファイルはアイコンで表現されます．ファイルに対する操作は，そのファイルのアイコンをマウスで操作することで実現します．

しかし本書ではファイル操作や処理の実行は，「シェル」と呼ばれるもので行います．シェルの操作はコマンド入力をベースにした「CUI」です．つまり「ファイル」は「ファイル名」をキーボードから入力してやることで指定します．

2.1 CUIでのコマンド操作に慣れよう

ＰＣの操作といえば，マウスを操作してアイコンをクリックするというものが一般的です．このような操作方法はGUI（Graphical User Interface）と呼ばれます．

しかし例えば前章では，ユーザ名の確認には「whoami」，スーパーユーザになるには「su」，パスワード変更には「passwd」，シャットダウンには「init 0」というコマンドをキーボードから入力して実行することで，操作を行いました．このような操作方法はCUI（Character User Interface）と呼ばれます．

今どきのＰＣでは，ユーザインタフェースはGUIが主流です．しかし本書では以下の理由から，あえてCUIをベースにして説明します．

- **GUIは流行の移り変わりが速く，変化するたびに新しく覚える必要が出てしまう．CUIは一度マスターしてしまえば，より長く利用できる．**
- **GUIになっているシステムもそのベースにはCUIがあり，GUIの裏側でCUIを実行している場合が多い．このためいざというときにCUIが役立つことも多い．**
- **サーバなどの操作はCUIで行うことが前提になっている場合があり，覚えておいて損は無い．**
- **コマンドの自動実行などを行う場合に，CUIのほうが相性が良くやりやすい．**

言うならば，GUIは初心者向け，CUIはエキスパート向け・プログラマ向けのユーザインターフェー

スとも考えられるでしょう．

ただ何事も，体験しておくことは良いことです．本書ではエキスパートになることを目標にはせずに，あくまで「体験」として，軽い気持ちでCUIに触れてみます．

2.1.1 ファイルの一覧を表示させよう

まずはVM上のCentOSを起動して，「user」でログインします．ログインすると以下のようなプロンプトが出てくるはずです．

```
[user@localhost ~]$
```

これは「シェル」というツールが起動し，コマンド入力を求めている状態です．つまりこれは「シェルのプロンプト」なわけです．シェルはCUIをベースにしたツールの代表格です．

ここで「ls」と入力して，Enterを押してみましょう．入力するのは英小文字で「エル」「エス」です．「l」は数字の「１」ではないので注意してください．

```
[user@localhost ~]$ ls
[user@localhost ~]$
```

これで「ls」というコマンドが実行されたことになります．「ls」は「list」の略で，ファイル一覧を表示するためのコマンドです．

しかし，上の例では何も表示されていません．これは，まだ何のファイルも作成されていないためです．

「touch」というコマンドで，空のファイルを作成することができます．試しに「sample.txt」というファイルを作成してみましょう．

```
[user@localhost ~]$ touch sample.txt
[user@localhost ~]$
```

もう一度，lsコマンドを実行してみましょう．

```
[user@localhost ~]$ ls
sample.txt
[user@localhost ~]$
```

今度は「sample.txt」というファイルが表示されました.

lsコマンドは，引数にファイルを指定することで，そのファイルがあることを確認することができます.

lsの引数に「sample.txt」を指定して，実行してみましょう.

```
[user@localhost ~]$ ls sample.txt
sample.txt
[user@localhost ~]$
```

このように，コマンドはその後にパラメータを指定することができます．このようなものを「コマンド引数」「コマンドライン引数」などと呼びます.

シェルではコマンドライン引数は，上のように空白で区切って指定します．空白はスペースキーで入力できます．スペースキーはキーボードの最下段中央にある，横長のキーです.

存在しないファイルを指定した場合にはどうなるのでしょうか．試しに「noexist.txt」というファイルを引数に指定して，lsコマンドを実行してみましょう.

```
[user@localhost ~]$ ls noexist.txt
ls: cannot access noexist.txt: No such file or directory
[user@localhost ~]$
```

「そのようなファイルは無い」として，エラーになっているようです.

またコマンドにはオプションを付けることができます．lsコマンドには「-l」（数字の「1」ではなく，英小文字のエルなので注意してください）というオプションがあり，ファイルの詳細情報を表示できます．やってみましょう.

```
[user@localhost ~]$ ls -l
total 0
-rw-rw-r--. 1 user user 0 Jun 22 16:00 sample.txt
[user@localhost ~]$
```

いくつかの情報が並んで表示されています．これは左から，ファイルの属性，リンク数（とりあえず気にしないでください），ファイルの所有ユーザ名，ファイルのグループ名，ファイルサイズ，ファイルの更新日付，そしてファイル名になります.

ファイル指定と「-l」オプションを組み合わせて使うこともできます．

```
[user@localhost ~]$ ls -l sample.txt
-rw-rw-r--. 1 user user 0 Jun 22 16:00 sample.txt
[user@localhost ~]$
```

2.1.2 入力間違いを修正しよう

ここでコマンドラインの操作について，説明しておきましょう．

コマンドラインには，入力した文字列がそのまま表示されます．例えば（これはよくある打ち間違いなのですが）「ls」と間違えて「sl」と入力してしまったとしましょう．

```
[user@localhost ~]$ sl
```

これでEnterを押すと，slというコマンドは存在しないため，そのようなコマンドは無いと言われてしまいます．

```
[user@localhost ~]$ sl
-bash: sl: command not found
[user@localhost ~]$
```

では，どのようにすれば入力した文字列を修正できるのでしょうか？

もう一度，「sl」と打ち間違えてしまったとしましょう．画面には次ページの**図2.1**のように表示されています．カーソルは「sl」の右側に表示されています．

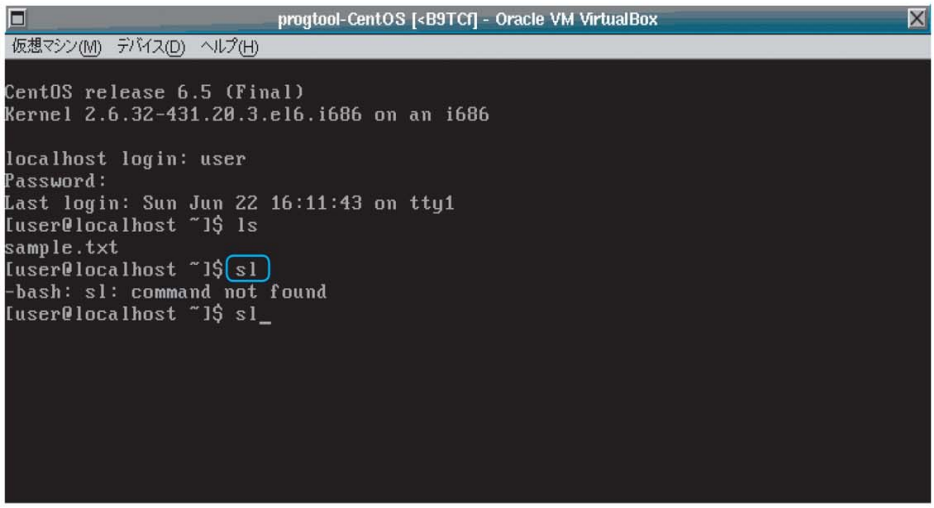


図2.1:「sl」と打ち間違えてしまった

「Backspaceキー」を押すことで，直前に入力した文字を削除できます．「Backspaceキー」はキーボードの右上あたりにありますので，まずは1回，押してみましょう．

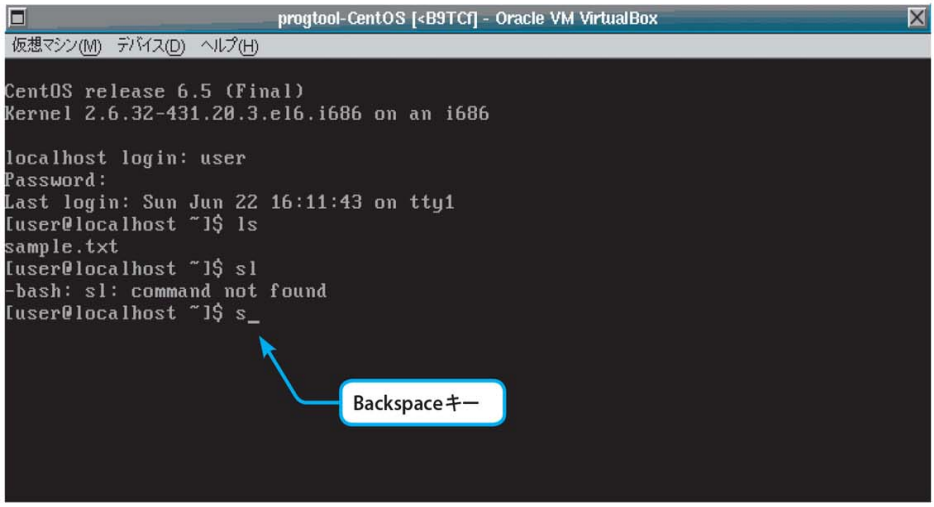


図2.2: Backspaceキーを1度押してみる

すると**図2.2**のようにコマンドラインの右端の「l」が削除され，カーソルがひとつ左に移動しています．このようにBackspaceキーを押すと，カーソルの直前の1文字が削除されます．

次に，矢印キーの「←」を押してみます．すると**図2.3**のようにカーソルが1文字ぶん左に動き，「s」の後ろから「s」の位置に移動します．

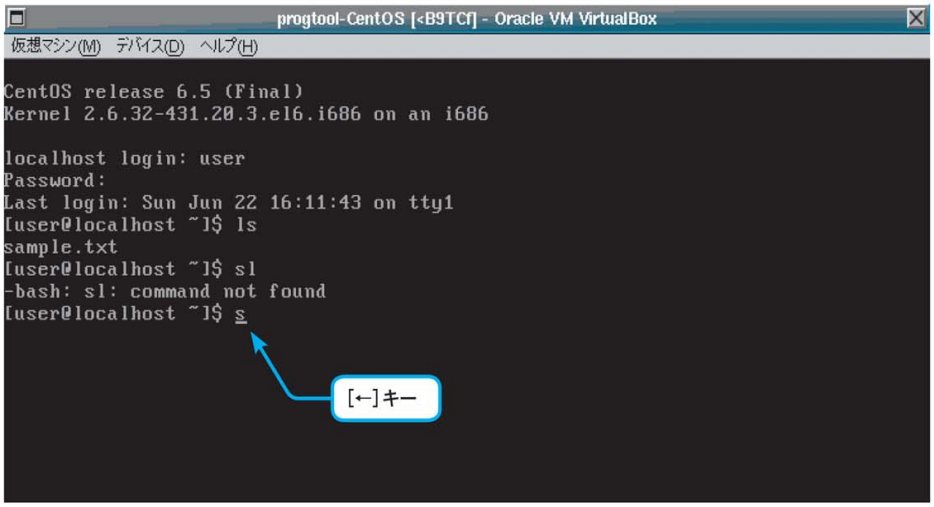


図2.3: 矢印キーの「←」を押す

この状態で「l」を押すとどうなるでしょうか？

入力した文字は，カーソルの位置に挿入されます．このため「l」を入力すると，**図2.4**のように「s」の前の位置に「l」を挿入することができます．

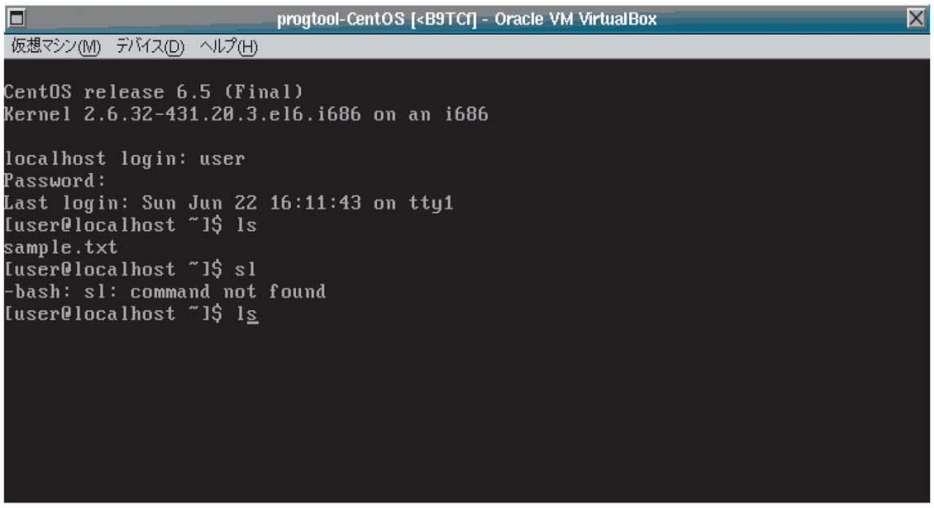


図2.4:「s」の前に「l」を挿入する

これでEnterを押せば，「ls」が実行されます．

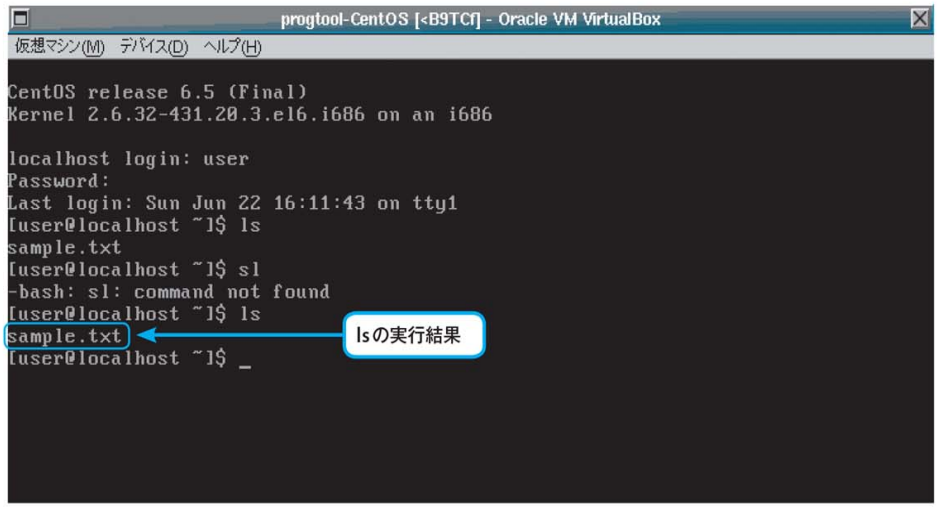


図2.5:「ls」が実行される

今度はうまく実行できました.

2.1.3 コマンドラインを編集してみよう

コマンドライン操作について，もう少し詳しく説明します.

例えば「abcdefg」と入力してみましょう．画面は**図2.6**のようになります.

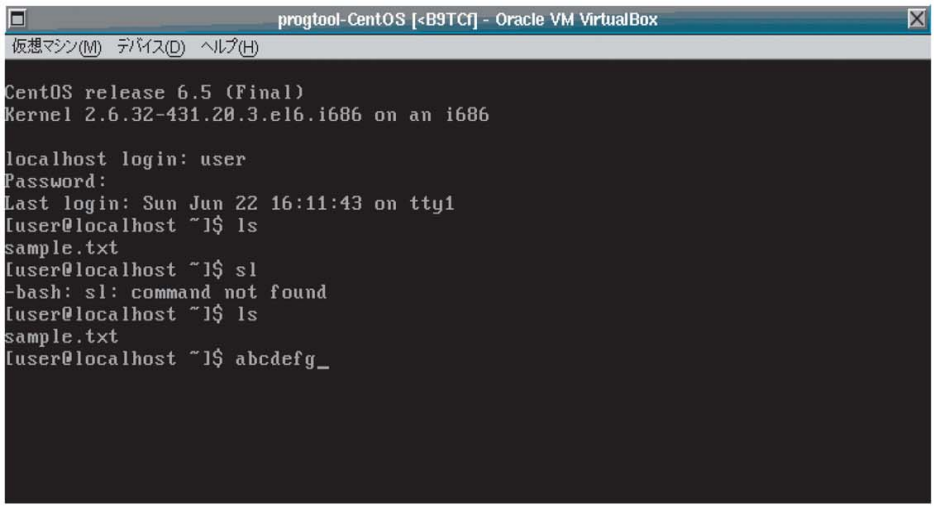


図2.6:「abcdefg」と入力する

コマンドラインの文字列は，「矢印キー」や「Backspaceキー」を用いて編集することができます．例えば「abcdefg」と入力した後に，「Backspaceキー」を２回，押してみます．

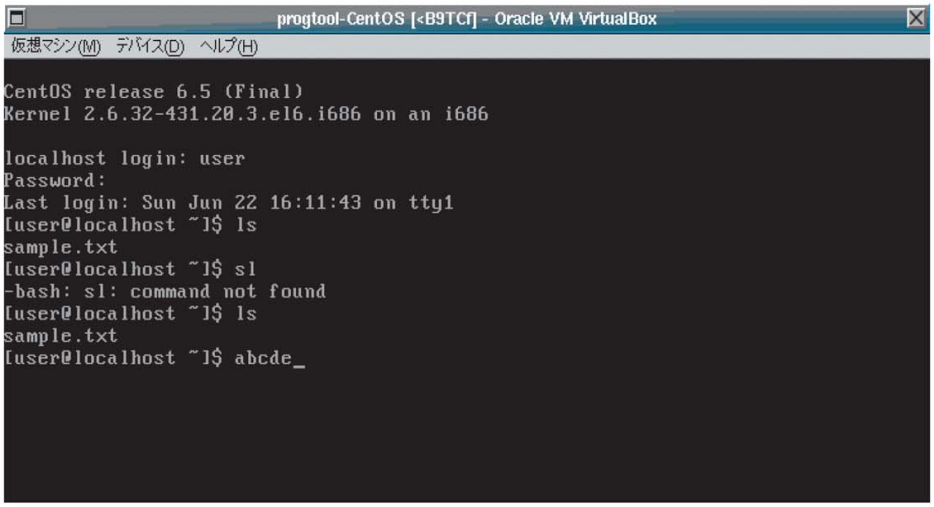


図2.7: Backspaceキーを２回押して「fg」を削除する

画面は**図2.7**のようになりました．「g」と「f」の文字が消え，カーソルは「e」の終端に戻っています．

ここで「ABC」と入力してみましょう．なおアルファベットの大文字は，「Shiftキー」を押しながらアルファベットキーを押すことで入力できます．「Shiftキー」はキーボードの右端と左端にひとつずつありますが，どちらを使っても構いません．使いやすいほうを使うといいでしょう．

例えば「A」のキーをそのまま押すと小文字の「a」が入力されますが，Shiftキーを押しながら「A」を押すと，大文字の「A」が入力されます．

つまり「ABC」を入力するためには，右端か左端のどちらかのShiftキーを押しながら「A」「B」「C」とキーを押します．すると**図2.8**のようになります．

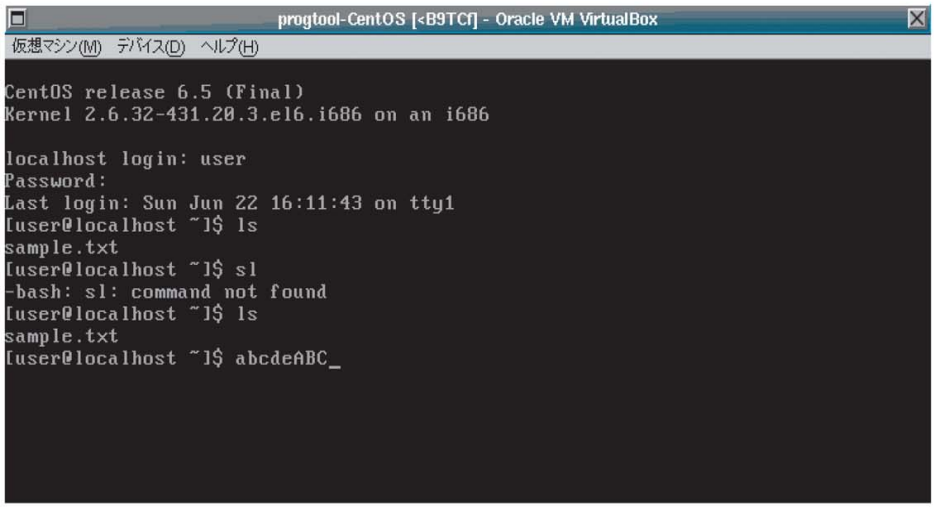


図2.8:「ABC」と入力する

次に矢印キーの「←」を5回，押してみましょう．するとカーソルが**図2.9**のように「d」の位置に移動するでしょう．

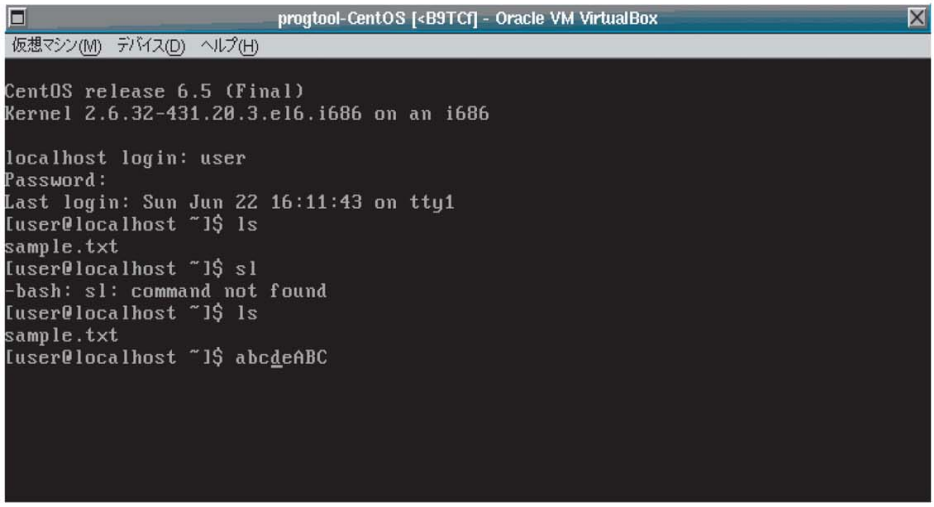


図2.9: 矢印キーの「←」を5回押す

ここで「DE」を入力してみます．**図2.10**のようになったでしょうか．

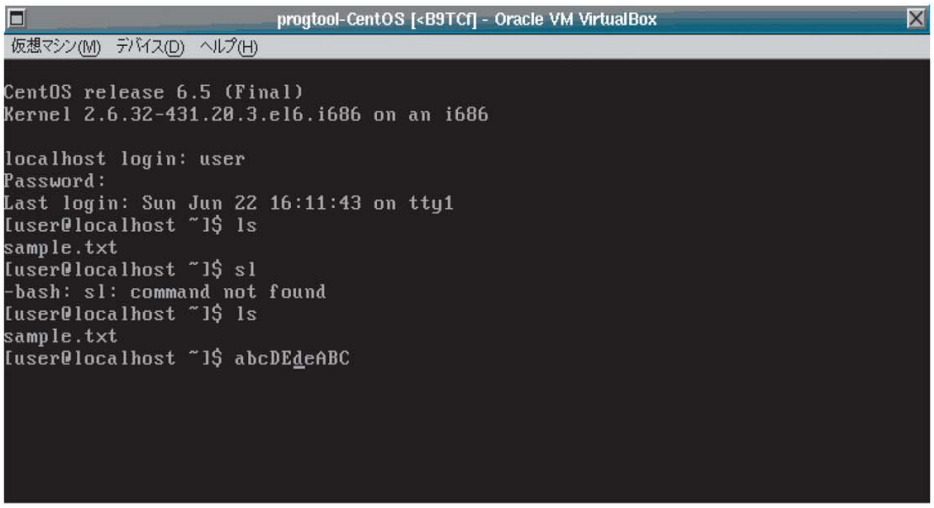


図2.10:「DE」と入力する

このように矢印キーでカーソルを移動して，カーソルの位置に文字を挿入することができます．

さらにBackspaceキーを1回，押してみましょう．

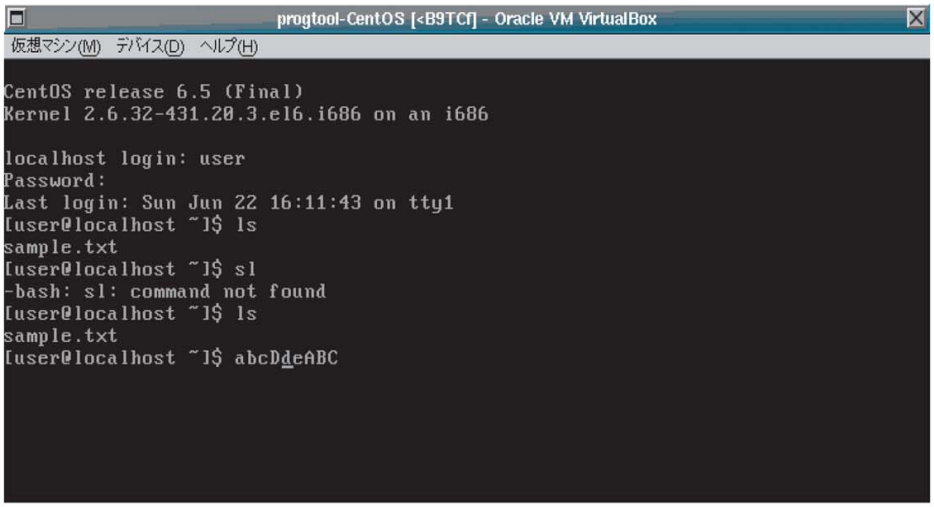


図2.11:「E」を削除する

図2.11のようになり，先ほど入力した「E」が削除されました．

2.1.4 実行の履歴を遡ってみよう

さて，CUIでは実行したいコマンドをいちいち入力する必要があるのですが，毎回これでは面倒です．CUIでのコマンド操作は，一見すると非常に面倒なものにも見えてしまうでしょう．

しかしシェルには，コマンド入力をサポートする様々な機能が用意されています．例えばコマンド実行の履歴を遡って，再度実行することができます．

矢印キーの「↑」を押すことで，直前に実行したコマンドが表示されます．**図2.12**は「ls」の実行後に「↑」を押したときの画面です．事前に実行した「ls」が，履歴としてコマンドラインに表示されています．

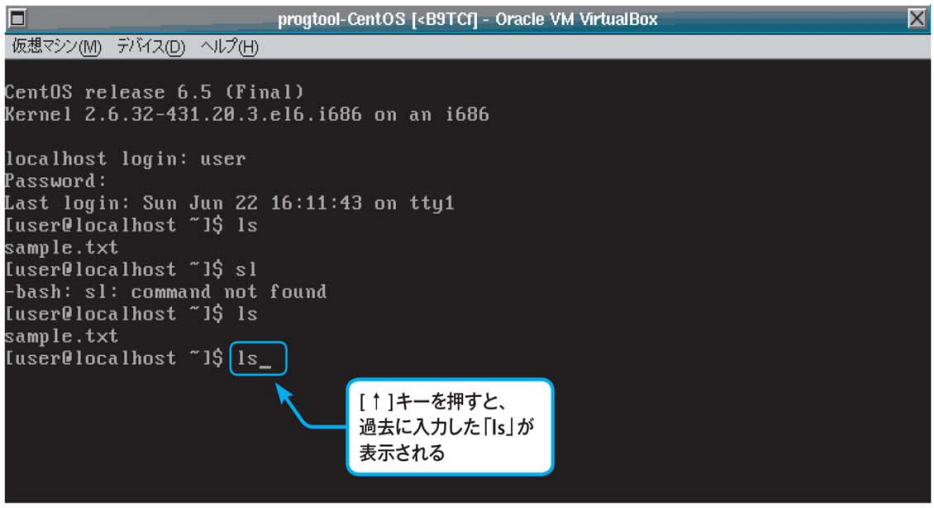


図2.12: 矢印キーの「↑」で履歴を遡る

ここでEnterを押すと，コマンドライン上に表示されている「ls」を実行できます．

「↑」を複数回押すと，実行したコマンドが，履歴を遡って順番に表示されます．また「↓」を押すと，戻る方向に表示されます．つまり過去に実行したコマンドならば，履歴表示させて気軽に再実行できるわけです．

図2.13は**図2.12**の状態で，もう一度「↑」を押したときの画面です．

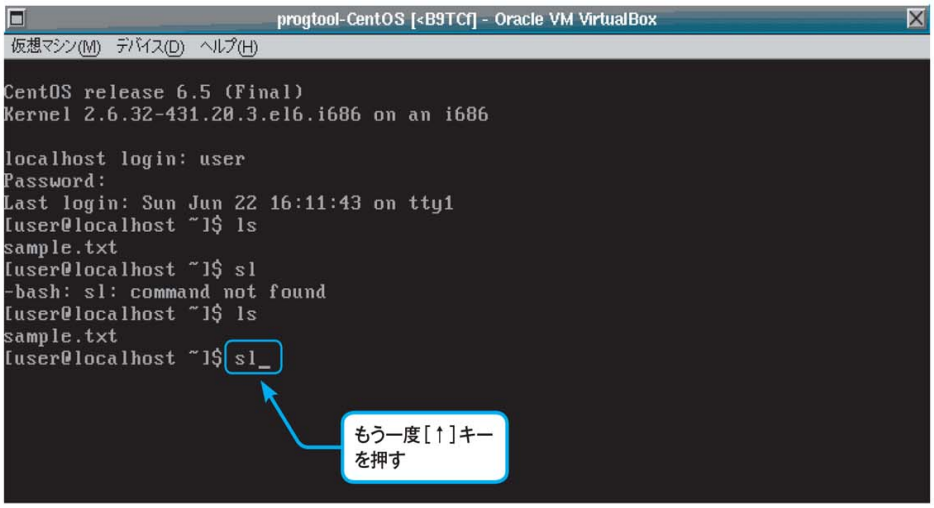


図2.13: もう一度「↑」を押す

図2.13ではlsの前に実行された「sl」というコマンドが履歴として表示されています．
これは実際にはslというコマンドは無く，以前に実行しようとしたときにはエラーになっていましたが，履歴には残っているようです．

過去のコマンド履歴が表示された状態で「←」やBackspaceキーを押せば，表示された内容を編集することもできます．そしてEnterを押せばコマンドが実行できます．

つまり「↑」によって履歴を遡って「sl」と表示させ，それを編集して実行させることもできるわけです．

2.1.5 補間機能を利用してみよう

次にコマンドの補間を試してみましょう.

例えば「whoami」を実行したいとき，まず**図2.14**のように「whoa」まで入力したとします.

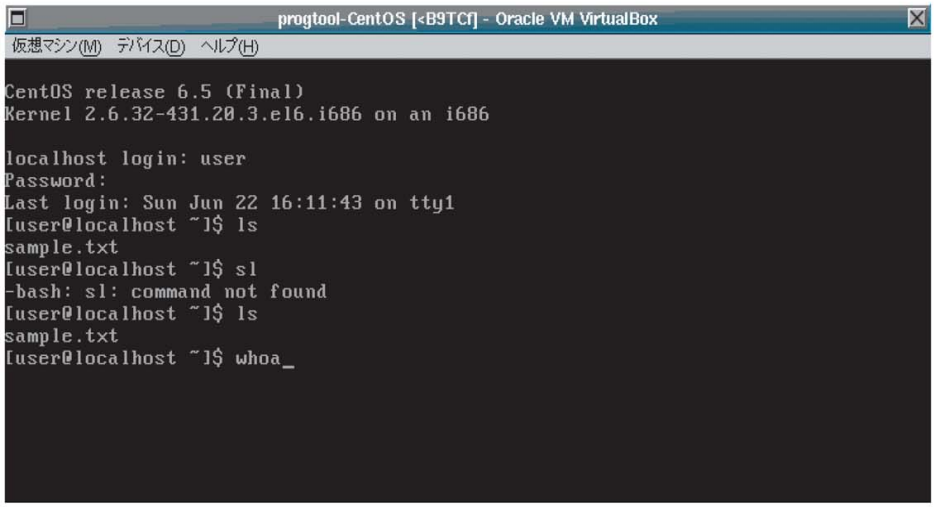


図2.14:「whoa」まで入力する

この状態で「Tabキー」を押してみましょう． Tabキーはキーボードの左端にあります.

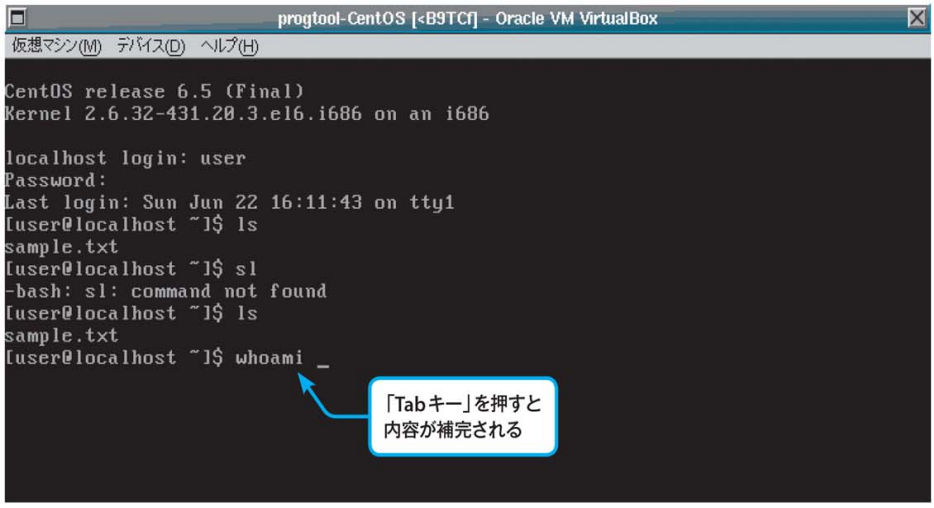


図2.15: Tabキーを押すと「mi」が補間される

Tabキーを押すと入力した「whoa」に続けてシェルが「mi」を補間し，**図2.15**のように「whoami」になりました．

ここでEnterキーを押せば，もちろん「whoami」として実行できます．

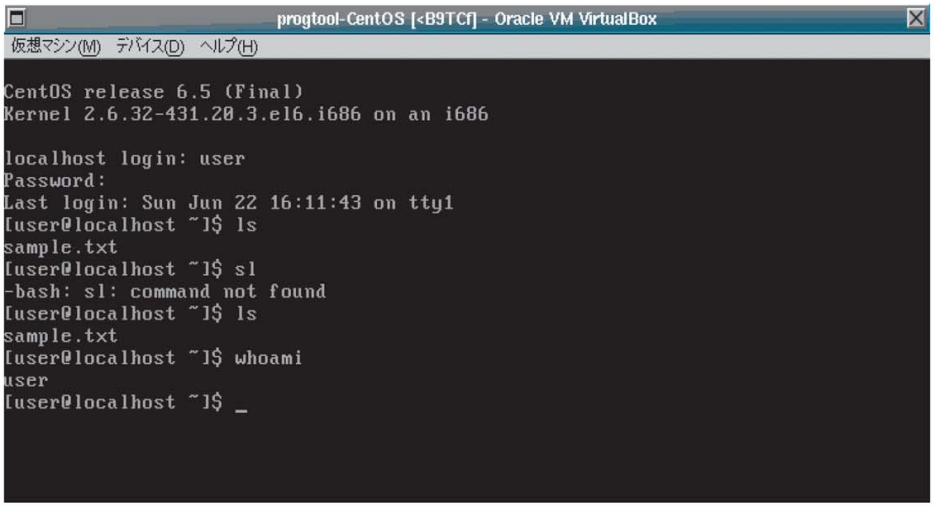


図2.16: 補間したコマンドを実行する

Tabキーを押すと，そこまで入力した文字列に対して当てはまるコマンドを補間します．

ここで要注意なのは，当てはまるコマンドが複数ある場合には補間できず候補が表示されるということです．例えば「who」だけを入力してTabキーを押しても，**図2.17**のように何も補間されません．

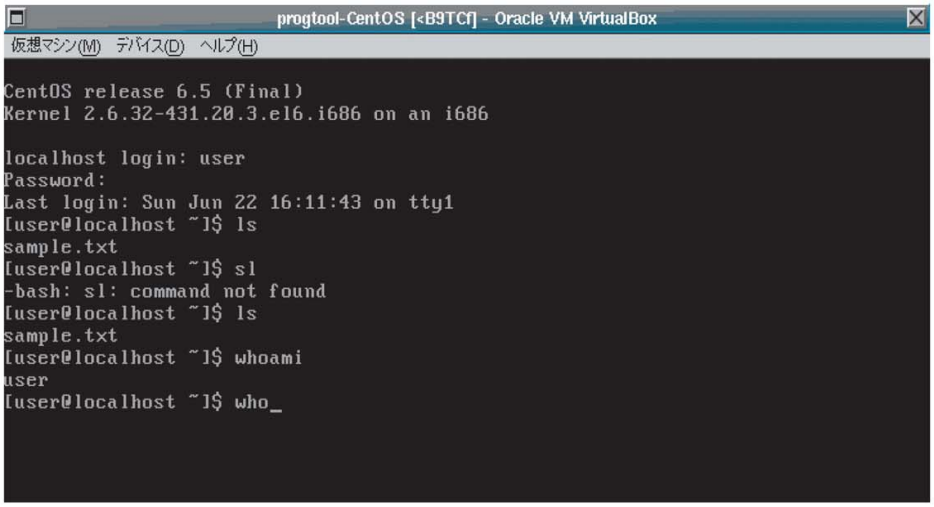


図2.17: 補間の候補が複数ある場合には，何も補間されない

Tabキーを一度押しただけでは何も起きないのですが，もう一度押すと**図2.18**のように補間の候補が表示されます．

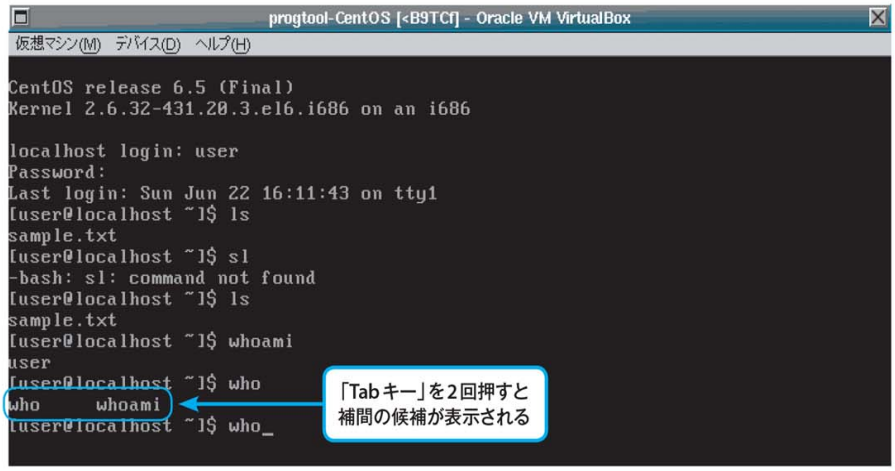


図2.18: Tabキーを２回押すと，補間の候補が表示される

図2.18では候補として「who」と「whoami」の２つが表示されました．つまり「whoami」の他に「who」というコマンドもあるため，「who」だけ入力した状態では補間の候補が複数あり，何を補間すればいいかを確定できないわけです．

「a」を入力して，もう一度Tabキーを押してみましょう．

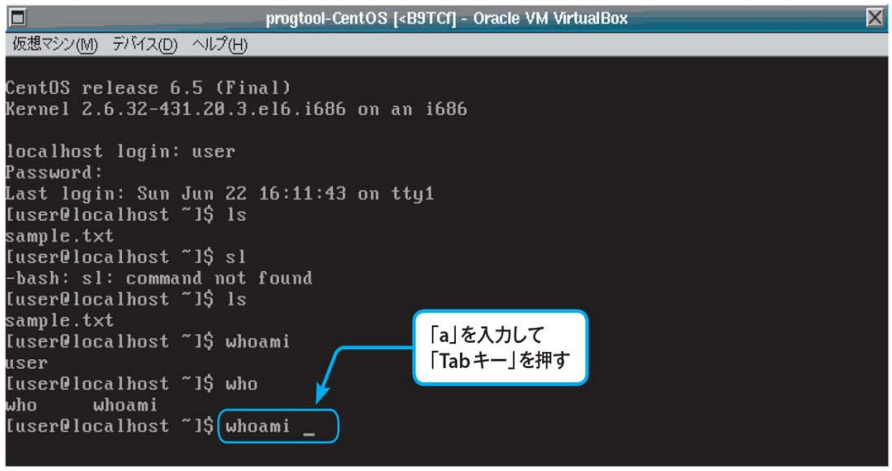


図2.19:「a」を入力後にTabキーを押すと，補間されて「whoami」になる

今度は補間され，**図2.19**のように「whoami」と表示されました．

2.2 ファイルを操作してみよう

ここまでで簡単なコマンドラインの操作方法を体験しました．基本はこれで十分ですので，あとはいろいろなコマンドを覚えていくだけです．

では，よく利用されるコマンドを，いろいろ試しつつ覚えていきましょう．

2.2.1 ファイルの中身を見てみよう

まずファイルの内容の参照です．

Windowsではファイルの内容は，ファイルのアイコンをダブルクリックすることでメモ帳などが開き，そこで見ることができます．

しかしCUIでは，ファイルの内容を参照するのももちろんコマンドです．「cat」というコマンドがありますので，それで見てみましょう．

```
[user@localhost ~]$ ls
sample.txt
[user@localhost ~]$ cat sample.txt
[user@localhost ~]$
```

現在は先ほどtouchコマンドで作成した「sample.txt」というファイルがあります．

しかしその内容を見ても，何も表示されていません．これはtouchコマンドで作成したため，空のファイルとなっているからです．

2.2.2 ファイルに書き込もう

ファイルへの書き込みには「echo」というコマンドが応用できます．

まず「echo」は，引数として与えられた文字列を単に表示するだけのコマンドです．

```
[user@localhost ~]$ echo SAMPLE
SAMPLE
[user@localhost ~]$
```

さらにシェルには，「リダイレクト」という機能があります．これはコマンドの後に「>」をつけることで，指定したファイルにコマンドの出力結果を書き込む，という意味になります．

例えば以下を実行してみましょう．

```
[user@localhost ~]$ echo SAMPLE > sample.txt
[user@localhost ~]$
```

今度は「SAMPLE」と表示されていません．これはechoコマンドの出力である「SAMPLE」という文字列が，sample.txtというファイルに書き込まれているためです．

catコマンドで中身を見てみましょう．

```
[user@localhost ~]$ cat sample.txt
SAMPLE
[user@localhost ~]$
```

確かにsample.txtに「SAMPLE」と書き込まれているようです．

リダイレクトの記号に「>」を使うと，ファイルへの書き込みは上書きになります．
例えば以下を実行してみましょう．

```
[user@localhost ~]$ echo test > sample.txt
[user@localhost ~]$ cat sample.txt
test
[user@localhost ~]$
```

先ほどの「SAMPLE」という文字列は上書きされ，「test」に内容が書き変わっています．

そして「>>」を使うと，追記という意味になります．

```
[user@localhost ~]$ echo sample2 >> sample.txt
[user@localhost ~]$ cat sample.txt
test
sample2
[user@localhost ~]$
```

今度は「sample2」という文字列が，「test」の後に追記されています．

リダイレクトはwhoamiやlsコマンドなどにも応用できます．

```
[user@localhost ~]$ whoami > sample.txt
[user@localhost ~]$ cat sample.txt
user
[user@localhost ~]$
```

もちろん，追記もできます．

```
[user@localhost ~]$ ls >> sample.txt
[user@localhost ~]$ cat sample.txt
user
sample.txt
[user@localhost ~]$
```

2.2.3 ディレクトリを作成しよう

次に「フォルダ」の作成です．

「フォルダ」は要するにファイルをまとめるための「箱」です．Windowsではマウスで右クリック後に「新規作成」-「新規フォルダの作成」のようにメニュー選択することでフォルダが作成できます．

「フォルダ」は「ディレクトリ」などとも呼ばれます．ということで本書では，「ディレクトリ」と呼ぶことにします．

しかしCUIでは，ディレクトリの作成もコマンドで行います．これには「mkdir」というコマンドを使います．

やってみましょう．引数には作成するディレクトリの名前を指定します．

```
[user@localhost ~]$ mkdir dir
[user@localhost ~]$
```

これで「dir」というディレクトリが作成されました．なお「mkdir」は「make directory」の略です．

作成したディレクトリを見てみましょう．

```
[user@localhost ~]$ ls
dir  sample.txt
[user@localhost ~]$
```

sample.txtに加えて，先ほど作成したディレクトリ「dir」が表示されています．

次に，ディレクトリの中に入ってみます．Windowsではディレクトリのアイコンをダブルクリックすることでその中に入れますが，シェルでは「cd」というコマンドでディレクトリ移動ができます．「cd」は「change directory」の略です．

```
[user@localhost ~]$ cd dir
[user@localhost dir]$
```

「dir」というディレクトリの中に入り，プロンプトに「dir」の文字列が追加されています．実はプロンプトには，現在いるディレクトリが表示されているのです．

ファイル一覧を見てみましょう．

```
[user@localhost dir]$ ls
[user@localhost dir]$
```

こんどは何も表示されません．作成した「dirディレクトリ」には，まだ何もファイルが無いためです．

今いるディレクトリのことを「カレントディレクトリ」と呼びます．つまり現在のカレントディレクトリは「dir」である，ということができます．

また，ログイン時の最初にいるディレクトリは「ホームディレクトリ」と呼びます．これはWindowsでいうところの「デスクトップ画面」に相当します．

2.2.4 ディレクトリ内にファイルを作成しよう

「touchコマンド」でdir内にファイルを作成してみましょう．

```
[user@localhost dir]$ touch sample2.txt
[user@localhost dir]$
```

これで「sample2.txt」という名前のファイルが作成されました．ファイル一覧はどうなるでしょうか．

```
[user@localhost dir]$ ls
sample2.txt
[user@localhost dir]$
```

「sample2.txt」というファイルが作成されています．

「sample.txt」は表示されていません．これは現在「dir」というディレクトリ内にいて，ディレクトリの外にあるファイルは表示されないためです．なおディレクトリ内にディレクトリを作成することもできます．

さて，現在は「dir」というディレクトリの中にいますが，元の場所に戻ってみましょう．ディレクトリ名に「..」を指定すると，ひとつ前のディレクトリを示すことになります．そこに「cd」することで，前のディレクトリに戻ることができます．

```
[user@localhost dir]$ cd ..
[user@localhost ~]$ ls
dir  sample.txt
[user@localhost ~]$
```

元のディレクトリに戻りました．「sample2.txt」はdirの中にあるため，今度はlsで表示されていません．

dirディレクトリに「sample2.txt」というファイルがあることを，もう一度確認してみます．lsコマンドに続けてディレクトリ名を指定することで，そのディレクトリの中のファイルの一覧を表示することができます．

```
[user@localhost ~]$ ls dir
sample2.txt
[user@localhost ~]$
```

2.2.5 ファイルをコピーしよう

ファイルはコピーしたり，ディレクトリ間で移動したりすることができます．

コピーは「cp」というコマンドで可能です．やってみましょう．まず，sample.txtの内容を確認します．

```
[user@localhost ~]$ cat sample.txt
user
sample.txt
[user@localhost ~]$
```

先ほどのリダイレクトの実験の結果が残っているようです．これをsample3.txtというファイルにコピーしてみましょう．

```
[user@localhost ~]$ cp sample.txt sample3.txt
[user@localhost ~]$
```

これで「sample.txt」というファイルをコピーして「sample3.txt」というファイルを作成します．

```
[user@localhost ~]$ ls
dir  sample3.txt  sample.txt
[user@localhost ~]$
```

コピーされたファイルの内容を見てみましょう．

```
[user@localhost ~]$ cat sample3.txt
user
sample.txt
[user@localhost ~]$
```

sample.txtの内容と一致しています．確かに，sample.txtの内容がコピーされてsample3.txtが作成されているようです．

2.2.6 ファイルを移動しよう

ファイルの移動は「mv」というコマンドで行えます．dirディレクトリにある「sample2.txt」というファイルを，こちらに移動してみましょう．

```
[user@localhost ~]$ mv dir/sample2.txt .
[user@localhost ~]$
```

これは今までに無いファイル指定方法なので，ちょっと説明しましょう．

まずディレクトリ内にあるファイルは，「ディレクトリ名/ファイル名」というように「/」（スラッシュ）で区切って指定できます．つまり「dir/sample2.txt」は，「dir」というディレクトリ内にある「sample2.txt」というファイルを指します．ディレクトリ内にディレクトリがあるような場合には，dir/dir2/sample2.txtのように連続して指定することもできます．

さらに移動先にディレクトリ名を指定すると，そのディレクトリ内にファイルを移動するという意味になります．そして「.」というディレクトリ指定は，カレントディレクトリを指します．

ということで「mv dir/sample2.txt .」というコマンドは，「dirディレクトリ内にあるsample2.txtというファイルを，カレントディレクトリに移動する」という意味になります．カレントディレクトリは現在はホームディレクトリになっていますから，結果的にsample2.txtをホームディレクト

リに移動することになります．

移動後にファイル一覧を見てみましょう．

```
[user@localhost ~]$ ls
dir  sample2.txt  sample3.txt  sample.txt
[user@localhost ~]$
```

sample2.txt が移動していることがわかります．dir内はどうなっているでしょうか．

```
[user@localhost ~]$ ls dir
[user@localhost ~]$
```

こちらは空になっています．mvコマンドにより，sample2.txt がホームディレクトリに移動したためです．

2.2.7 複数のファイルを一度に扱ってみよう

ファイルのコピーや移動はできるようになりましたが，例えば「sample.txt」と「sample2.txt」という２つのファイルを移動したい場合には，どうしたらいいでしょうか？

もちろん，以下のように２回に分けて移動することは可能です．

```
[user@localhost ~]$ mv sample.txt dir
[user@localhost ~]$ mv sample2.txt dir
```

そしてこれは，実は以下のように１回で実行することもできます．

```
[user@localhost ~]$ mv sample.txt sample2.txt dir
```

しかしいずれにしても，移動するファイルをすべて指定しなければならないことは同じです．

これはファイルが２個や３個のときにはいいのですが，ファイルが５個くらいになると面倒になってきます． 10個だと，ちょっとやりたくない作業です． 100個だと，途中で諦めてしまいそうな数になってしまいます．

こういうときに便利なものとして，「ワイルドカード」というファイルの指定方法がシェルにはあります．例えば「sample1.txt」と「sample2.txt」の２つのファイルは，コマンドライン上では「sample*.txt」のように表現することができます．

実際に，やってみましょう．まずlsコマンドで，現在あるファイルを確認します．

```
[user@localhost ~]$ ls
dir  sample2.txt  sample3.txt  sample.txt
[user@localhost ~]$ ls dir
[user@localhost ~]$
```

「sample.txt」「sample2.txt」「sample3.txt」という３つのファイルと，「dir」というディレクトリがあります．

ここで「mv sample*.txt dir」を実行してみましょう．

```
[user@localhost ~]$ mv sample*.txt dir
[user@localhost ~]$ ls
dir
[user@localhost ~]$ ls dir
sample2.txt  sample3.txt  sample.txt
[user@localhost ~]$
```

「sample.txt」「sample2.txt」「sample3.txt」の３つのファイルが，dirに一気に移動されています．

コマンドライン上では，「*」は「すべての文字列」に当てはまります．つまり「sample*.txt」と書けば，「sample.txt」「sample2.txt」「sample3.txt」のすべてを示すことになります．「空の文字列」にも当てはまるため，「sample.txt」にも当てはまります．

つまり上のコマンドは，以下を実行するのと等価になるわけです．

```
[user@localhost ~]$ mv sample.txt sample2.txt sample3.txt dir
```

ワイルドカードは他にもあります．例えば「?」は，「任意の１文字」に当てはまります．

試してみましょう．まず比較用に，新たにファイルを追加します．

```
[user@localhost ~]$ touch dir/sample123.txt
[user@localhost ~]$ ls dir
sample123.txt  sample2.txt  sample3.txt  sample.txt
[user@localhost ~]$
```

これらを移動してみます．まずは「*」を使った移動です．

```
[user@localhost ~]$ mv dir/sample*.txt .
[user@localhost ~]$ ls
dir  sample123.txt  sample2.txt  sample3.txt  sample.txt
[user@localhost ~]$ ls dir
[user@localhost ~]$
```

「*」はすべての文字列に当てはまります．このため「sample123.txt」にも当てはまるので，「sample123.txt」も移動されています．

次に「?」を試してみましょう．

```
[user@localhost ~]$ mv sample?.txt dir
[user@localhost ~]$ ls
dir  sample123.txt  sample.txt
[user@localhost ~]$ ls dir
sample2.txt  sample3.txt
[user@localhost ~]$
```

今度は「sample2.txt」と「sample3.txt」の２つだけがdir内に移動されて，「sample.txt」と「sample123.txt」は残っています．

「?」は任意の「１文字」に当てはまります．このため「sample2.txt」や「sample3.txt」には当てはまるのですが，「sample.txt」や「sample123.txt」には当てはまりません．よってこれらのファイルは移動されずに，残っているわけです．

2.2.8 拡張子って何だろう?

ところで今までファイルを「sample.txt」や「sample2.txt」のようなファイル名で作成してきました．このファイル名を見て，「sample」の部分は「ああ，サンプルだからなんだな」と想像がつきます．

しかしその後の「.txt」という部分は何なのでしょうか？

ファイルには好きな名前をつけることができます．たとえば「.txt」という部分を省いて，「sample」というファイル名でファイルを作成することもできます．

しかし通常は，後ろにそのファイルの種類を示す文字列をピリオドで区切って１～４文字くらいで追加します．このような文字列を「拡張子」と呼びます．

以下は拡張子の例です．

- **「.txt」… テキストファイル**
- **「.zip」… ZIP圧縮したファイル**
- **「.jpg」… JPEG画像のファイル（「.jpeg」としたりもします）**
- **「.html」… HTML文書のファイル**

また実行ファイルは拡張子無しとします．たとえばここまでで何度か利用している「ls」というコマンドは，実は「/bin」というディレクトリに実行ファイルがあります．見てみましょう．なお「/bin/…」のように先頭に「/」を付加すると，先頭のディレクトリ（ルートディレクトリと言います）という意味になります．

```
[user@localhost ~]$ ls /bin/ls
/bin/ls
[user@localhost ~]$
```

これらの約束ごとは，必ずしも守らなければならない規則というわけではありません．しかし約束を守ることで，ファイルの種類がひとめで判別できるようになり便利です．とくに理由が無いならば，なるべく守ったほうがいいでしょう．

2.2.9 ファイルを削除しよう

ここまでで試しにいくつかのファイルとディレクトリを作成してきましたが，最後に掃除のために，これらの削除方法も説明しておきましょう．

まずはファイルは「rm」というコマンドで削除できます．これは「remove」の略です．引数には削除したいファイルを指定します．

sample.txt を削除してみましょう．

```
[user@localhost ~]$ ls
dir  sample123.txt  sample.txt
[user@localhost ~]$ rm sample.txt
[user@localhost ~]$
```

削除に確認を求めてきた場合には「y」を入力してEnterを押します．

ファイルが削除されたことを確認してみましょう.

```
[user@localhost ~]$ ls
dir  sample123.txt
[user@localhost ~]$
```

sample.txt が削除されています．続けて，sample123.txtも削除してみます.

```
[user@localhost ~]$ rm sample123.txt
[user@localhost ~]$ ls
dir
[user@localhost ~]$
```

次はワイルドカードを使って削除してみます.

ただしワイルドカードを使った削除は，指定のしかたを間違えると消したくないファイルを消してしまったり，ファイルをごっそりと消してしまったりということもあり得るので，十分に注意して実行してください.

ここでは「dir/sample*.txt」のように指定して，dirディレクトリの中にあるファイルを削除してみます.

```
[user@localhost ~]$ ls dir
sample2.txt  sample3.txt
[user@localhost ~]$ rm dir/sample*.txt
[user@localhost ~]$ ls dir
[user@localhost ~]$
```

最後に，ディレクトリは「rmdir」というコマンドで削除できます． rmdirは「remove directory」の略です.

```
[user@localhost ~]$ rmdir dir
[user@localhost ~]$ ls
[user@localhost ~]$
```

ただしディレクトリ内にファイルが残っていると，そのディレクトリは削除できないので注意してください.

2.3 まとめ

本章ではシェルの操作を体験してみました．これは本書で，この後に様々なプログラムを書いたりツールを使ったりしていく上でのベース知識になります．

ＰＣの操作のほとんどは，実はファイルの操作です．このためファイルの基本操作ができないと，様々なツール類を使うこともおぼつかなくなってしまいます．ここでしっかりと押さえておきましょう．

3

さまざまなコマンドを覚えよう

[準備編3]—コマンド操作

前章ではシェルの基本操作について説明しましたが，次は応用編として，いくつかのツールの使いかたや組み合わせかたを説明します．

シェルによるコマンド操作は，さまざまなツールを知れば知るほど応用範囲が広がります．シェルは知れば知るほど，圧倒的に素早く操作ができるようになったり，やりたいことがシンプルに実現できるようになったりする，エキスパート向けのツールと言えます．

もっとも本書の目的は「体験すること」なので，そのようなエキスパート向けの話まではしません．ただある程度知っておくと，作業の効率が格段に上がることも事実ではあります．

本章ではそうした「これくらいは知っておくと便利な機能」を，いくつか追加で説明します．

3.1 ファイルをビューワで見てみよう

ファイルの内容はcatコマンドで参照できます．

しかしファイルの内容がそれなりのサイズになる場合には，catコマンドだとファイルの内容が下から上に流れてしまい，末尾のほうしか残りません．

例えば/usr/binというディレクトリには，実行できるコマンドがいろいろ置かれています．これはlsコマンドで見ることができます．

```
[user@localhost ~]$ ls /usr/bin > ls.txt
[user@localhost ~]$
```

しかしここで作成した「ls.txt」をcatコマンドで見てみても，**図3.1**のように画面に入りきらず，出力は一瞬にして画面外に流れていってしまいます．

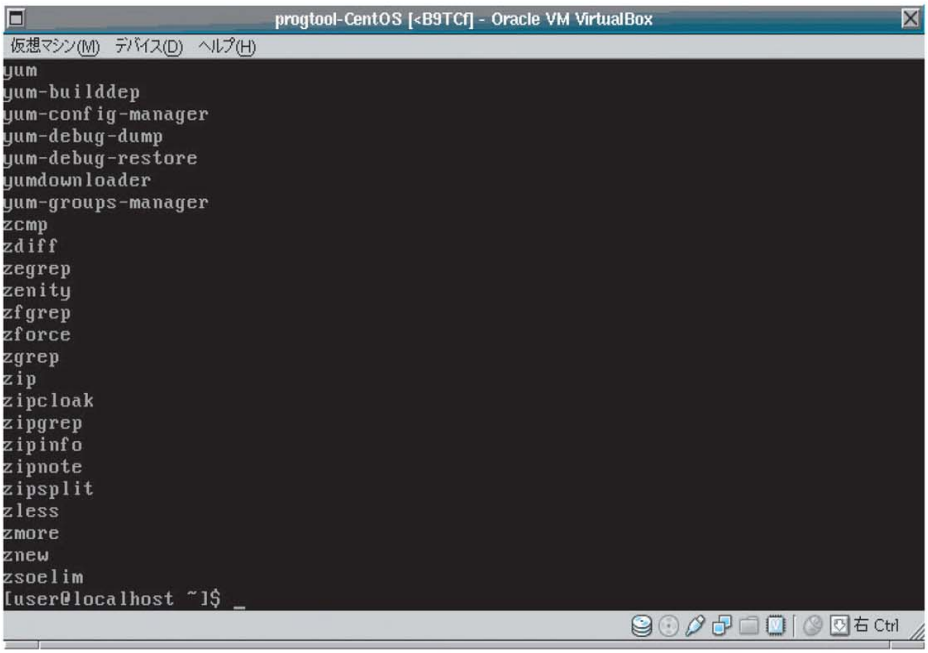


図3.1: 出力結果が入りきらず，画面外に流れていってしまう

3.1.1 lessコマンドを使ってみよう

「less」というファイルビューワのコマンドがありますので，これでファイルを参照することができます．

まず，以下のようにコマンドを実行してみてください．

```
[user@localhost ~]$ less ls.txt
```

実行すると**図3.2**のような画面になります．

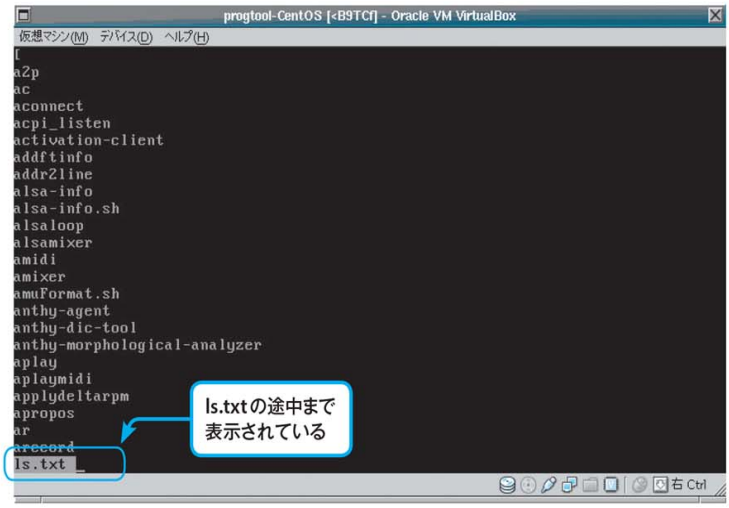


図 3.2: less でファイルの内容を表示する

あとは矢印キーの上下で，ファイルの内容を上下に移動しながら読むことができます．**図 3.3** は「↓」を押して，少し下のあたりを見てみたところです．

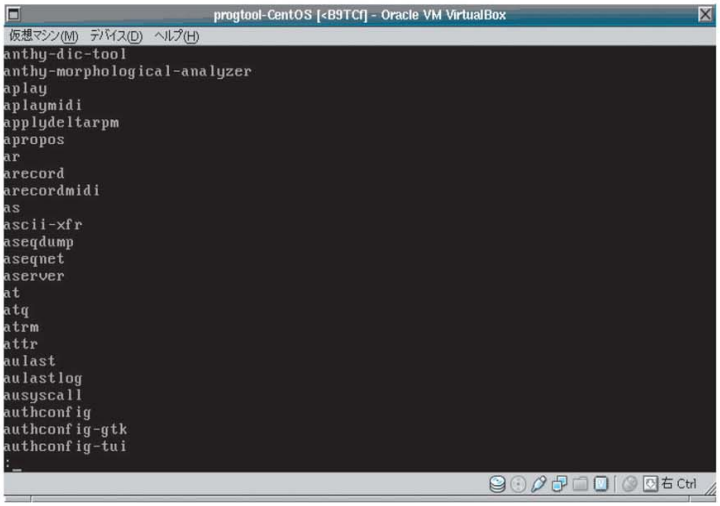


図 3.3: 矢印キーの上下でスクロールする

また「<」（「Shift」＋「<」のキー）を入力することで，ファイルの先頭に一気にジャンプします．「>」はファイルの末尾にジャンプします．

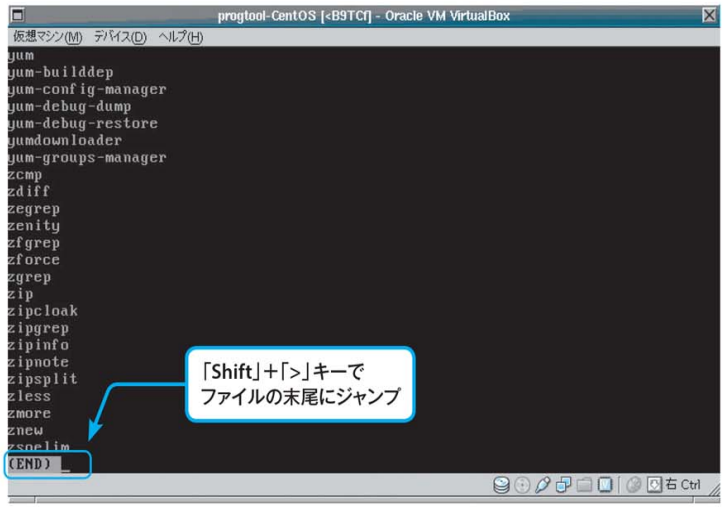


図3.4:「>」を押して，ファイルの末尾にジャンプする

3.1.2 単語を検索してみよう

lessコマンドでは，簡単な「検索」も行うことができます．まず「<」を押して，先頭に戻ってみましょう．画面は**図3.5**のようになります．

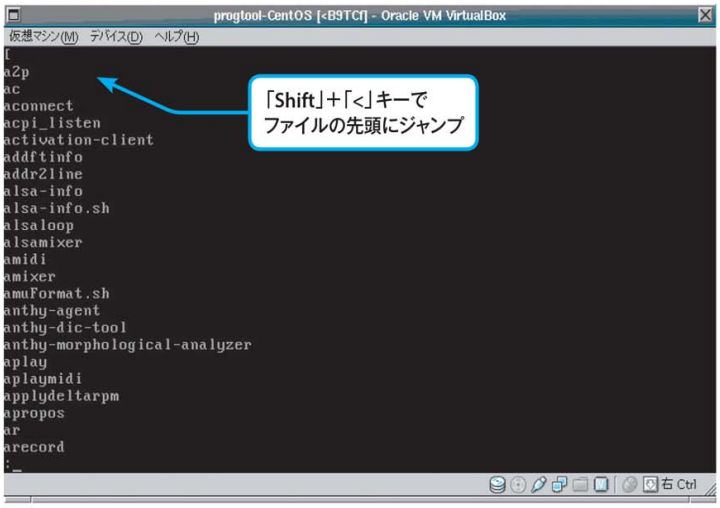


図3.5:「<」を押して，ファイルの先頭にジャンプする

次に「/」を入力してみましょう．キーボード右下あたりにある「/」のキーをそのまま押します．

すると**図3.6**のように，画面の左下に「/」が表示されます．

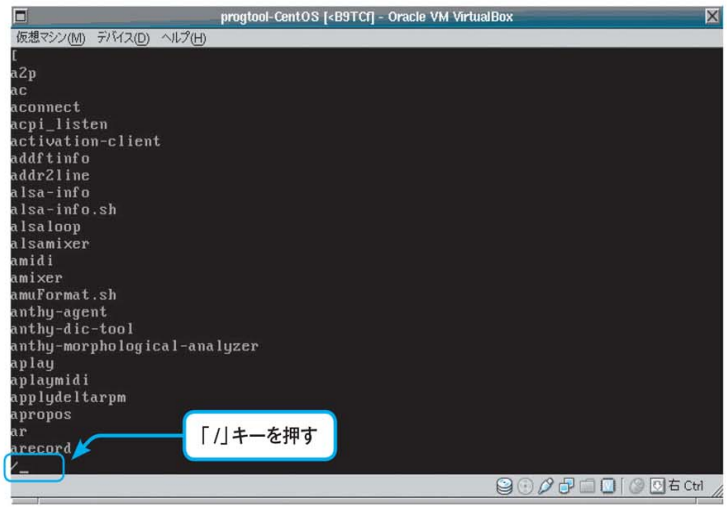


図3.6:「/」を押して検索モードに入る

ここで検索する文字列を入力します．例えば「passwd」と入力してみましょう．

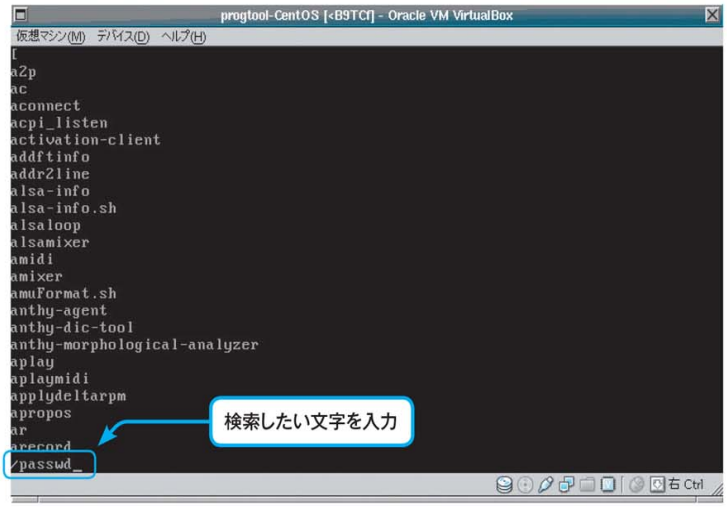


図3.7:「/」を押した後，「passwd」と入力する

Enter を押すと，検索開始です．

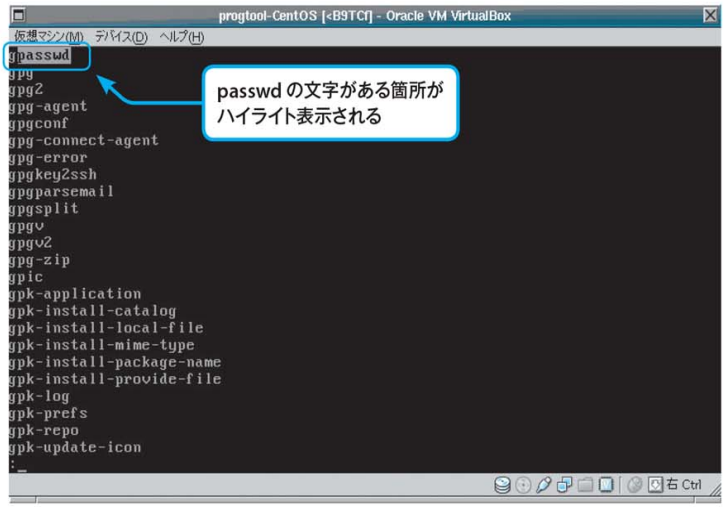


図3.8: Enterを押して検索を開始する

図3.8では「gpasswd」というコマンドがヒットして，「passwd」の部分がハイライトされています．このように行の中に部分的に検索文字列があれば，そこが検索にヒットします．

続けて次を検索する場合には，「/」を押してそのままEnterを押します．

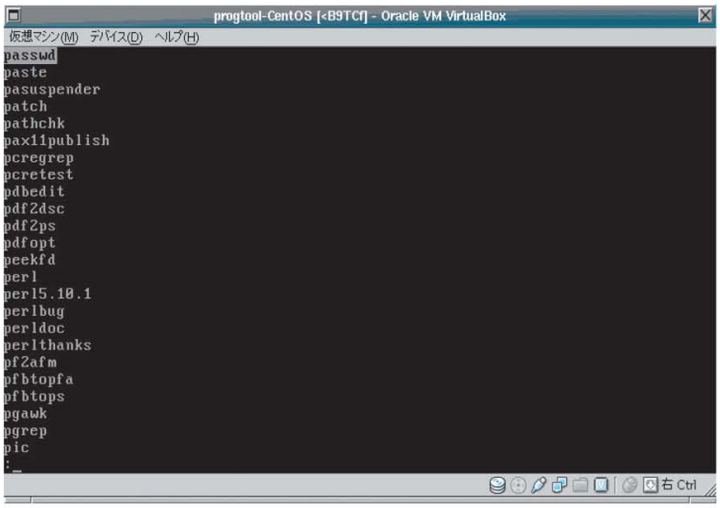


図3.9:「/」「Enter」で次を検索する

今度は**図3.9**のように，「passwd」という行が検索できました．これはpasswdコマンドの実行ファイルになります．

lessは「Q」を押すことで終了します．終了すると**図3.10**のようになります．一番下の行にプロンプトが出力されて，コマンドの入力待ちに戻っています．

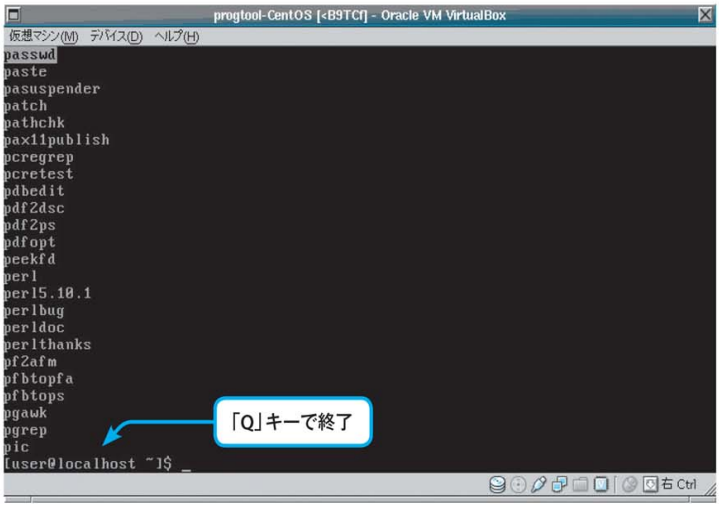


図3.10:「Q」で終了する

最後に，作成した「ls.txt」を消して掃除しておきましょう．

```
[user@localhost ~]$ rm ls.txt
[user@localhost ~]$
```

3.2 「リダイレクト」と「パイプ」を使ってみよう

シェルには「リダイレクト」と「パイプ」と呼ばれる，特徴的な機能があります．

これはコマンドが出力する内容をファイルに書き込んだり，別のコマンドの入力として受け渡すというものです．

これらの機能により，シェルの環境では複数のコマンドを連携させて動作させることができます．

3.2.1 リダイレクトを試してみよう

まずは「リダイレクト」という機能を試してみましょう．例えば，以下を実行してみてください．

```
[user@localhost ~]$ echo SAMPLE
SAMPLE
[user@localhost ~]$
```

これは前章でも説明しましたが，復習としてもう一度説明しましょう．

echoコマンドは引数として指定された文字列を出力するだけのコマンドです．上の例では引数に指定された「SAMPLE」という文字列を，そのまま出力しています．

そしてその出力は「>」としてファイル名を指定することで，ファイルに書き出すことができます．

```
[user@localhost ~]$ echo SAMPLE > sample.txt
[user@localhost ~]$ cat sample.txt
SAMPLE
[user@localhost ~]$
```

これは「echoコマンド」の出力結果を，「sample.txt」というファイルに書き込んでいます．このためcatでsample.txtの内容を出力すると，上のようにして「SAMPLE」というファイルの内容が表示されています．

echoコマンドはそのまま実行すれば画面上に表示されるわけですが，「>」を末尾に付加すると，その指す先のファイルに書き込むということになります．これを「リダイレクト」と呼びます．「リダイレクション」などと呼ばれることもあります．

これはechoコマンドに限った話ではなく，様々なコマンドに共通で使えます．例えばここまでで何度か利用している，「cat」というコマンドで試してみましょう．

まずcatコマンドは，実行時の引数としてファイルが指定されると，そのファイルを読み込んでそのまま出力します．

```
[user@localhost ~]$ cat sample.txt
SAMPLE
[user@localhost ~]$
```

実行時に末尾に「> sample2.txt」を付加すると，catコマンドの出力を画面に出さずに「sample2.txt」というファイルに書き込むという意味になります．つまりこれは，ファイルのコピーと同じになります．

```
[user@localhost ~]$ cat sample.txt > sample2.txt
[user@localhost ~]$ cat sample2.txt
SAMPLE
[user@localhost ~]$
```

catコマンドは引数を指定せずに実行すると「キーボードからの入力を読み込む」という動作になります．例えば以下を実行してみてください．

```
[user@localhost ~]$ cat > sample.txt
(ここでブロックする)
```

実行すると，コマンドが終了せずに固まったような状態になっていると思います．

でも，あせる必要はありません．キーボードから，何か入力してみましょう．

```
[user@localhost ~]$ cat > sample.txt
abcde
```

ここでEnterを押してから，「Ctrl」＋「D」を入力します．「Ctrl」＋「D」というのは，キーボードの左下あたりにある「Ctrl」というキーを押しながら「D」を押す，ということです．1章ではpasswdコマンドの中断のために「Ctrl」＋「C」を使いましたが，それと似たキーボード操作になります．

つまりキーボードを押す順番は，以下のようになります．

「A」→「B」→「C」→「D」→「E」→「Enter」→「Ctrl」→（Ctrlを押したまま）「D」

すると，以下のようにコマンドが終了してくれます．

```
[user@localhost ~]$ cat > sample.txt
abcde
[user@localhost ~]$
```

これで実は，キーボードから入力した内容がリダイレクトされ，sample.txtに書き込まれています．

sample.txtの内容を見てみましょう．

```
[user@localhost ~]$ cat sample.txt
abcde
[user@localhost ~]$
```

「abcde」が入力されたようです．このように，catコマンドはキーボードからの入力を受け付けてファイルを作成するために利用できます．

「>」はファイルへの出力という意味になりますが，「<」は指定したファイルから入力する，という意味になります．

例えば「cat」は，以下のように実行すると，指定されたファイルの内容を出力します．

```
[user@localhost ~]$ cat sample.txt
```

しかしこれは，以下のようにしてもファイルの内容を出力させることができます．

```
[user@localhost ~]$ cat < sample.txt
```

前者は引数を解釈して指定されたファイルを読み込むという，catコマンドの機能です．

それに対して後者は，catコマンドの入力にsample.txtを流し込むという，シェルの機能を利用しています．

3.2.2 パイプを試してみよう

もうひとつシェルの便利な機能として，「パイプ」というものを試してみましょう．

たとえば次ページのコマンドを実行すると，出力数が多すぎて画面内には収まらず，末尾以外は一瞬で画面から流れて消えてしまいます．

```
[user@localhost ~]$ ls /usr/bin
```

このようなときは「headコマンド」を利用することで，先頭のみを表示できます．

以下を試してみましょう．

```
[user@localhost ~]$ ls /usr/bin | head -n 10
[
a2p
ac
aconnect
acpi_listen
activation-client
addftinfo
addr2line
alsa-info
alsa-info.sh
[user@localhost ~]$
```

先頭10行だけが表示されています．

「|」は「パイプ」と呼ばれるシェルの機能で，コマンドの出力を別のコマンドの入力に繋げるという意味になります．これはcatコマンドによりファイルの内容を出力し，その出力を画面に表示するのではなく，「head」というコマンドの入力に接続することになります．

そして「head」は，入力された内容の先頭数行のみを出力するというコマンドです．ここでは「-n 10」のように指定されているので，先頭10行のみが表示されることになります．

「head」に対して「tail」というコマンドもあります．こちらは末尾の数行を出力します．やってみましょう．

```
[user@localhost ~]$ ls /usr/bin | tail -n 10
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
zless
zmore
znew
zsoelim
[user@localhost ~]$
```

ここまでだと，リダイレクトとパイプのメリットはよくわからないかもしれません．

しかしこれらは，様々なコマンドを組み合わせることで威力を発揮します．コマンドについては後で説明しますので，まずはやりかたを覚えてしまってください．

3.2.3 標準入力と標準出力って何だろう?

さて，ここまでで「キーボードからの入力」「画面に出力」などと説明してきました．

例えば今までの説明のとおりに考えると，以下はキーボードからの入力をそのまま画面に出力する，という動作をします．

```
[user@localhost ~]$ cat
```

実際にやってみましょう．

```
[user@localhost ~]$ cat
abcde（ここでEnterを押す）
abcde（さらにCtrl+Dを押す）
[user@localhost ~]$
```

キーボードから「abcde」と入力しEnterを押すと，画面にそのまま「abcde」と出てきます．「Ctrl」＋「D」を押すと終了します．

ここで「キーボードからの入力」と言っていますが，これは「標準入力」と呼ばれます．上のcatコマンドの動作は正しく言うと，「標準入力からの入力を受け取り，それをそのまま標準出力に出力する」ということになります．

そしてリダイレクトの「<」が付加されていると，指定したファイルを標準入力に接続します．「>」が付加されていると，標準出力にやはりファイルを接続します．「|」が付加されていると，前のコマンドの標準出力を次のコマンドの標準入力に接続します．

結果として指定されたファイルの内容をcatコマンドで読み込んだり，catコマンドの出力がファイルに書き出されたり，コマンドの出力をheadコマンドで先頭のみ表示したりすることになるわけです．

3.3 色々なコマンドを使ってみよう

ここまででシェルの操作方法はだいたいマスターできました.

あとはコマンドを徐々に覚えていくだけです. シェルの環境ではコマンドを組み合わせて利用することができるため，コマンドを覚えれば，その組み合わせ数に応じてできることが増えていきます.

3.3.1 grepで検索してみよう

grepは検索をするためのコマンドです. 指定したキーワードを含む行を抽出して出力してくれます.

例えば /usr/bin というディレクトリには，様々なコマンドの実行ファイルが置かれています.

```
[user@localhost ~]$ ls /usr/bin
```

これでは，大量のファイルが表示されてしまいました.

この中から「passwd」という文字列を含む実行ファイルだけを抽出してみましょう.

```
[user@localhost ~]$ ls /usr/bin | grep passwd
gpasswd
passwd
smbpasswd
userpasswd
vncpasswd
[user@localhost ~]$
```

上の例ではlsコマンドの出力をパイプでgrepに流し込み，「passwd」を含む行を検索して出力しています． grepコマンドはこのようにパイプによって他のコマンドと組み合わせて，様々に応用することができます.

先頭であることを示すには，「^」を付加します．例えば以下のようにすると，「a」で始まる行のみを出力させることができます．

```
[user@localhost ~]$ ls /usr/bin | grep ^a
a2p
ac
aconnect
... (中略) ...
authconfig-tui
auvirt
awk
[user@localhost ~]$
```

思いの他，たくさん表示されてしまいました．このようなときは head と組み合わせることもできます．例えば先頭５行だけ，表示させてみましょう．

```
[user@localhost ~]$ ls /usr/bin | grep ^a | head -n 5
a2p
ac
aconnect
acpi_listen
activation-client
[user@localhost ~]$
```

上のようにして，コマンドの出力をパイプで次のコマンドに渡し，その出力をさらにパイプで別のコマンドに渡すこともできます．このようにすれば，grep した結果をさらに head に渡すことで，先頭の数個だけを見ることができるわけです．

もしくはファイルビューワである「less」と組み合わせて，見てみることもできます．

```
[user@localhost ~]$ ls /usr/bin | grep ^a | less
```

やはりパイプを複数回利用して，今度は less コマンドに出力を流し込んでいます．このようにすれば検索結果を上下にスクロールしながら，ゆっくり調べることもできます．

3.3.2 sortでソートしてみよう

sortは名前のとおりソート，つまり順番に並べ替えをするコマンドです．

例えばcatコマンドを使って，以下のようなファイルを用意してみます．

```
[user@localhost ~]$ cat > names.txt
Yamada
Kawada
Shimada
[user@localhost ~]$
```

catコマンドの実行後はキーボードからの入力待ちになりますので，上のように「Yamada」や「Kawada」などの名前を入力してはEnterを押していきます．最後の名前を入力してEnterを押した後，「Ctrl」＋「D」で入力を終了します．

これらをABC順に並べ替えてみましょう．

```
[user@localhost ~]$ cat names.txt | sort
Kawada
Shimada
Yamada
[user@localhost ~]$
```

ABC順に並び替えられています．

これは降順に並び替えられていますが，-rオプションを付加すると昇順になります．

```
[user@localhost ~]$ cat names.txt | sort -r
Yamada
Shimada
Kawada
[user@localhost ~]$
```

今度は順番が逆になっています．

数字のソートはどうでしょうか．まずは以下のようにcatコマンドで，適当な数字のファイルを作成します．入力が完了したら「Ctrl」＋「D」で終了してください．

```
[user@localhost ~]$ cat > numbers.txt
10
2
1
100
3
[user@localhost ~]$
```

これをsortコマンドに流し込むことで，ソートしてみましょう．

```
[user@localhost ~]$ cat numbers.txt | sort
1
10
100
2
3
[user@localhost ~]$
```

おや，おかしいですね．数値の小さい順番に並べかえられていません．

これは，数の値の順ではなくあくまで「辞書順」で並び替えられているためです．数を数値として扱うにはsortコマンドに「-n」というオプションを付加します．

```
[user@localhost ~]$ cat numbers.txt | sort -n
1
2
3
10
100
[user@localhost ~]$
```

今度は数値で降順に並び替えられました．

3.3.3 複数のコマンドを繋げてみよう

ここでちょっとしたコマンドの応用例を紹介しましょう．

例えば names.txt について，もう一度考え直してみます．

```
[user@localhost ~]$ cat names.txt
Yamada
Kawada
Shimada
[user@localhost ~]$
```

ここで names.txt は３つの名前しか書いてありませんが，ちょっと名前を追加してみましょう．

```
[user@localhost ~]$ cat >> names.txt
Honda
Kawada
Yamada
Kawada
[user@localhost ~]$
```

いくつかの名前は重複して，４つの名前を追加してみました．追加した後の names.txt は以下のようになります．

```
[user@localhost ~]$ cat names.txt
Yamada
Kawada
Shimada
Honda
Kawada
Yamada
Kawada
[user@localhost ~]$
```

names.txt には，重複した名前があります．例えば「Yamada」は２箇所に，「Kawada」は３箇所にあります．

ここで names.txt から名前の個数をカウントするにはどうすればいいでしょうか？　つまり，もしも「Yamada」が２箇所にあればそれはひとつとしてカウントする，ということです．

これは以下のようにして，１行のコマンドでカウントできます．

```
[user@localhost ~]$ cat names.txt | sort | uniq | wc -l
4
[user@localhost ~]$
```


なぜこれでうまくカウントできるのか，ちょっと説明しましょう．

まず重複する行は，「uniq」というコマンドで１行にまとめることができます．しかしuniqコマンドがまとめるのは「隣り合った重複行」です．よって２行の「Yamada」が離れた行にある場合，それらをまとめることはできません．

ここで「sort」が使えます．sortにより名前を並べ替えれば，同じ行は隣合わせになります．さらにuniqを通せば，隣り合った重複行として１行にまとめられるわけです．

```
[user@localhost ~]$ cat names.txt | sort
Honda
Kawada
Kawada
Kawada
Shimada
Yamada
Yamada
[user@localhost ~]$ cat names.txt | sort | uniq
Honda
Kawada
Shimada
Yamada
```

最後に「wc -l」はファイルの行数をカウントします．これにより行数，つまり名前の個数が出力されるというわけです．なお「-l」は小文字の「エル」です．数字の「１」ではないので注意してください．

```
[user@localhost ~]$ cat names.txt | sort | uniq | wc -l
4
[user@localhost ~]$
```

どうでしょうか．このようにいくつかの単純な動作をするコマンドをパイプによって複数接続し，様々に応用できるところが，シェルの便利なところです．

3.3.4 便利なマニュアル「manコマンド」を使おう

ここまでいくつかのコマンドの使いかたを説明してきましたが，しかしコマンドの細かい動作をすべて覚えておくことは難しいでしょう．コマンドリファレンスやマニュアルといった，ドキュメントがほしいところです．

実はこの「コマンドリファレンス」もコマンドになっています．それは「man」というコマンドです．

例えばsortコマンドには「-r」や「-n」というオプションがありましたが，他にはどんなオプションがあるのでしょうか．そんなときは以下を実行してみましょう．

```
[user@localhost ~]$ man sort
```

すると，**図3.11**のような画面になり，sortコマンドの説明が表示されます．

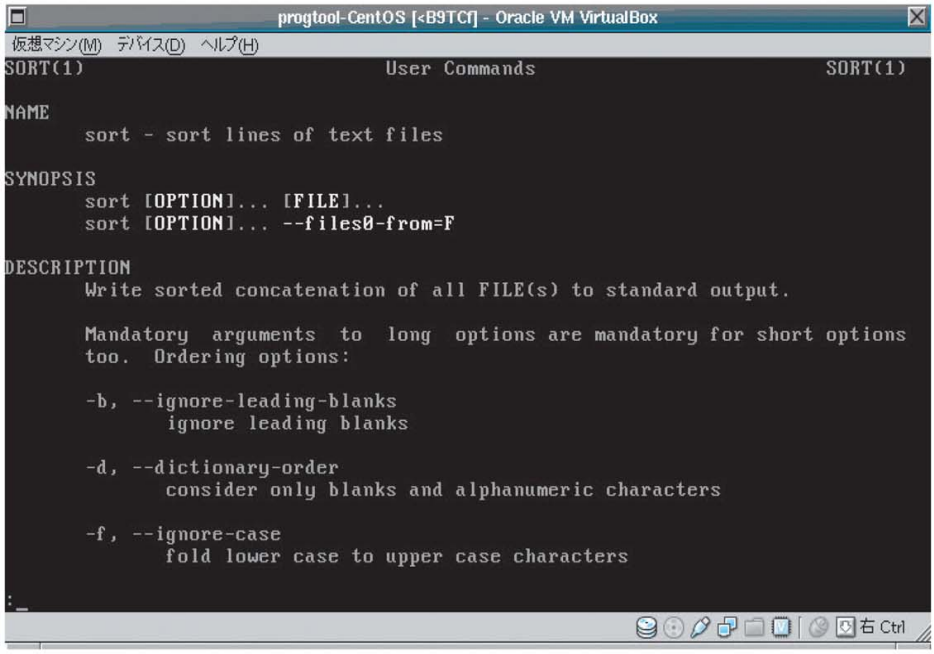


図3.11: man sortの実行画面

説明は矢印キーの上下でスクロールさせて読むことができます．

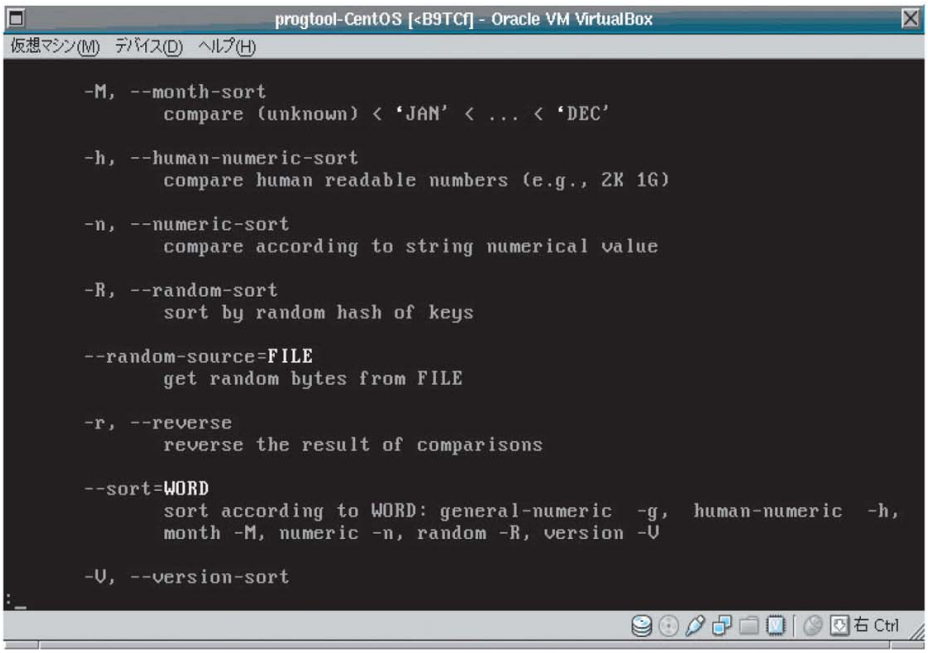


図3.12: sortコマンドのオプションの説明

見ると**図3.12**のような部分があり，「-n」や「-r」オプションの説明があります．他にも「-M」というオプションで，月表記（JAN, FEBなど）のソートもしてくれるようです．

そしてマニュアルは英文ですが，ポイントを押さえて読めば簡単です．他にももちろん，「man ls」や「man cat」もあります．コマンドのオプションなどがわからないときには，ぜひ活用してみましょう．

最後に，manは「Q」を押すと終了します．

3.4 まとめ

本章ではシェルの操作に加えて，ビューワや検索など，いくつかのツールの使いかたを説明しました．

シェルによるユーザインターフェースは，単純な動作をするコマンドを複数組み合わせて使える点に長所があります．これらのツールはひとつひとつは単純な機能しか持っていませんが，シェルの上で組み合わせて使うことで，様々に活用することができます．

つまりコマンドをたくさん覚えれば覚えるほど，それらが有機的に結び付いて，応用範囲が様々に広がっていくわけです．もしも興味が湧いたら，ぜひ積極的に覚えていくといいでしょう．

4

テキストエディタを使ってみよう
[ツール編 1]—nanoエディタ

シェルによるファイル操作を覚えた後は，テキストファイルの編集をマスターしましょう．

Windowsならば，テキストファイルは「メモ帳」や「ワードパッド」を使って自由に読み書きすることができます．このようなツールは「テキストエディタ」と呼ばれます．

テキストエディタの用途は，以下のようなものがあります．

- C言語などでプログラミングをする際に，プログラムを記述する
- 各種設定変更のときの，設定ファイルの書き換えをする

CentOSでは「nano」というシンプルなテキストエディタを追加インストールして使うことができます．

4.1 エディタ「nano」を使ってみよう

本書のVM上のCentOS環境には，すでにnanoがインストール済みです．

さっそくnanoを起動してみましょう．CentOSにログインしたら，以下を実行します．

```
[user@localhost ~]$ nano
```

すると**図4.1**のような画面になるはずです．

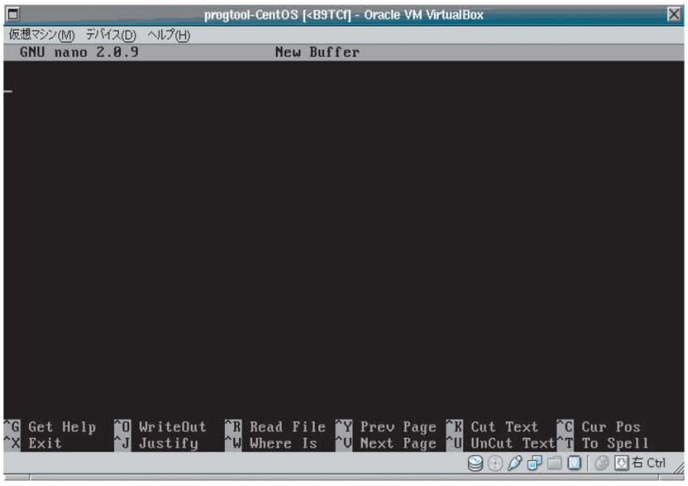


図4.1: nanoを起動する

4.1.1 文章を入力してみよう

図4.1の画面の左上にあるアンダーバーは，カーソルです．入力した文字は，カーソルの位置に挿入されていきます．

まずは適当な文章を書き込んでみましょう．キーボードで，「A」から「G」までのキーを順番に押してみます．

すると，**図4.2**のような画面になったはずです．

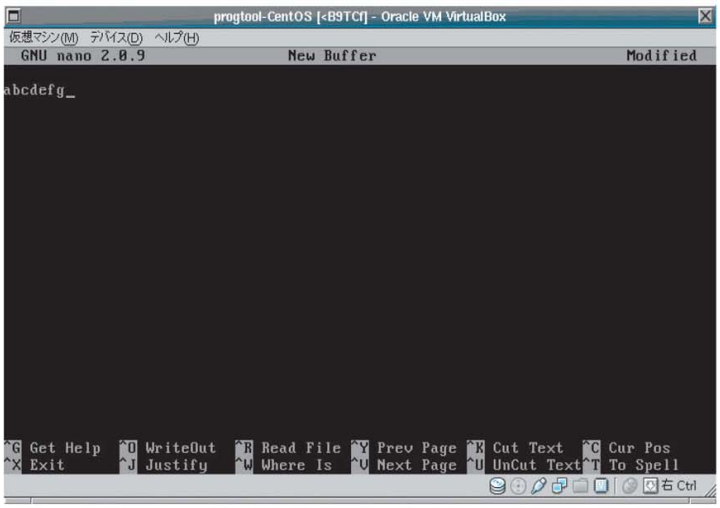


図4.2:「A」から「G」までを順番に押す

カーソルは入力に合わせて右に進み，**図4.2**では「g」の右隣にあります．

ここで矢印キーの「←」を数回，押してみましょう．

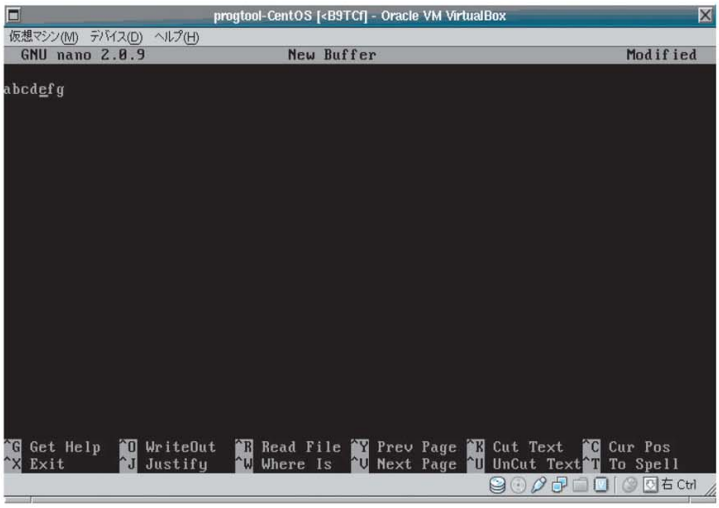


図4.3:「←」を3回押す

図4.3は「←」を3回押したところですが，カーソルが3つだけ左に戻り，「e」の位置に移動しました．

ここで，「ABC」と入力してみます．英字の大文字はShiftキーと英字キーをいっしょに押すことで入力できます．つまり，Shiftキーを押しながら「A」→「B」→「C」のようにキーを押していきます．

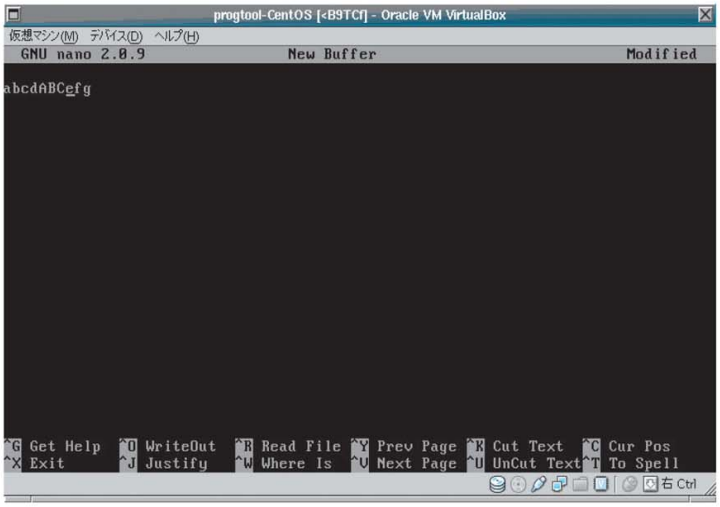


図4.4:「ABC」と入力する

すると**図4.4**のようになり，カーソルがあった「e」の位置に「ABC」の3文字が挿入されました．

文字の削除は「Backspace」のキーで可能です．試しに「Backspace」を３回，押してみます．

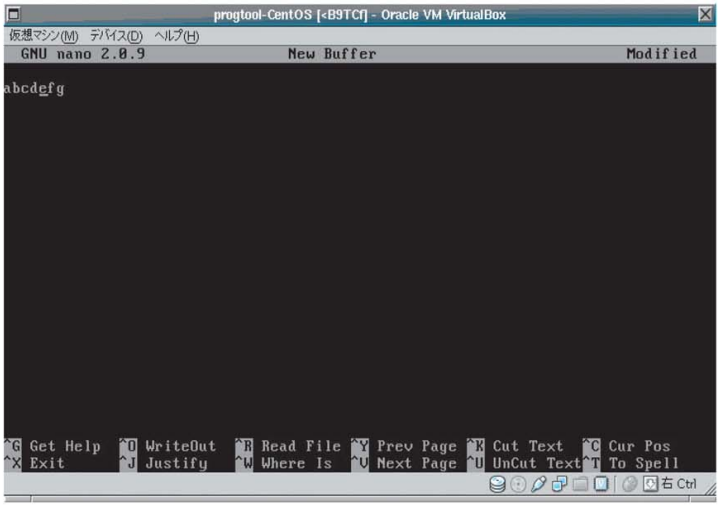


図4.5: Backspaceキーで文字を削除する

Backspaceを１回押すと，カーソルがひとつ左に移動して，そこにあった文字を削除します．**図4.5**では３回押すことで，先ほど入力した「ABC」の３文字が削除されました．

次に「Enter」を押してみます．

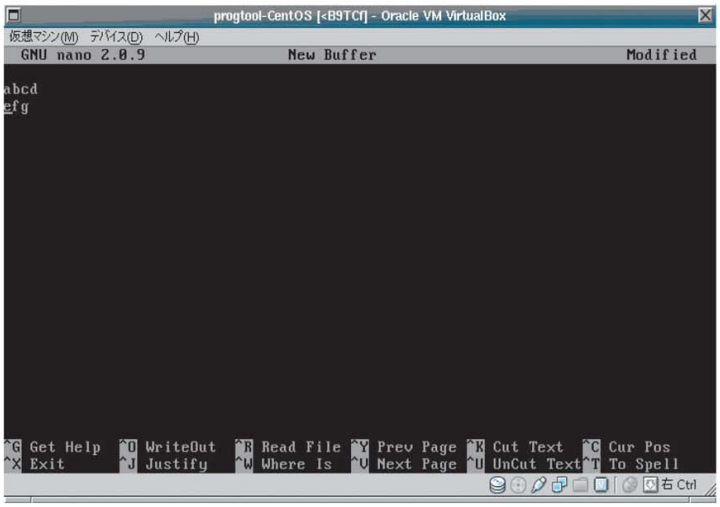


図4.6: Enterキーで改行を入力する

Enterキーを押すと，**図4.6**のようになります．「Enter」は改行になるので，「e」の位置で行が折り返されています．

Enterキーにより入力された改行も，１文字として扱われます．このため**図4.6**の状態で「Backspace」を押すと，「改行」の文字が削除され，**図4.7**の状態に戻ります．

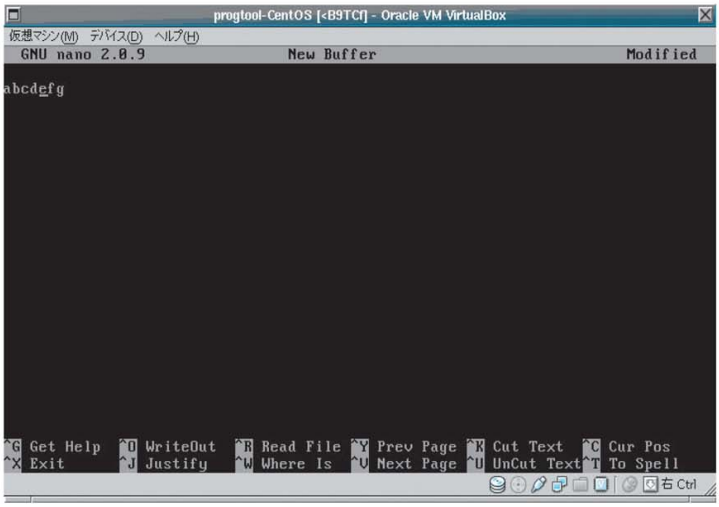


図4.7: Backspaceキーで改行文字を削除する

4.1.2 メニューの機能を実行してみよう

nanoの画面を見ると，画面下には「^G Get Help」や「^X Exit」といったコマンドのメニュー表示があります．

これはCtrlキーといっしょにキーを押すことで，当該の機能を実行することができます．例えば「^G Get Help」というのは，Ctrlキーを押しながら「G」を押すことで，ヘルプが表示されるということです．

「Ctrl」＋「G」を実際に押してみると，**図4.8**のようなヘルプ画面が表示されました．

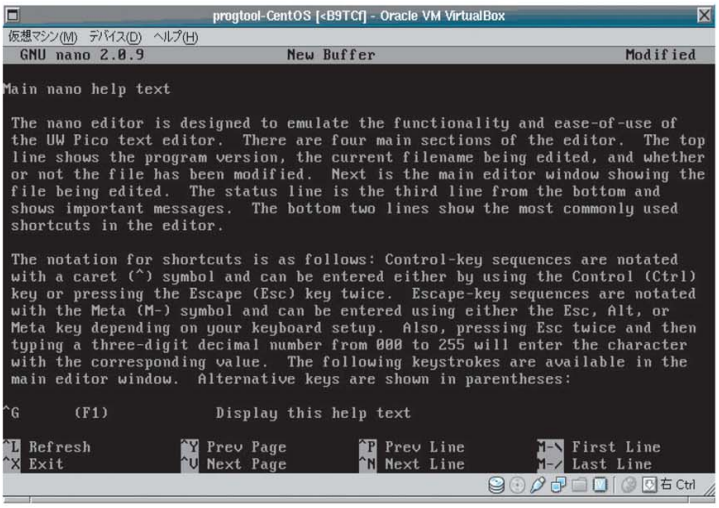


図4.8:「Ctrl」＋「G」でヘルプを表示する

ヘルプ画面の最下行にも「^X Exit」というメニューが出ていますので，ヘルプ画面は「Ctrl」＋「X」で終了できるようです．

実際に押すとこれでヘルプから抜けて，**図4.9**のような画面に戻ることができます．

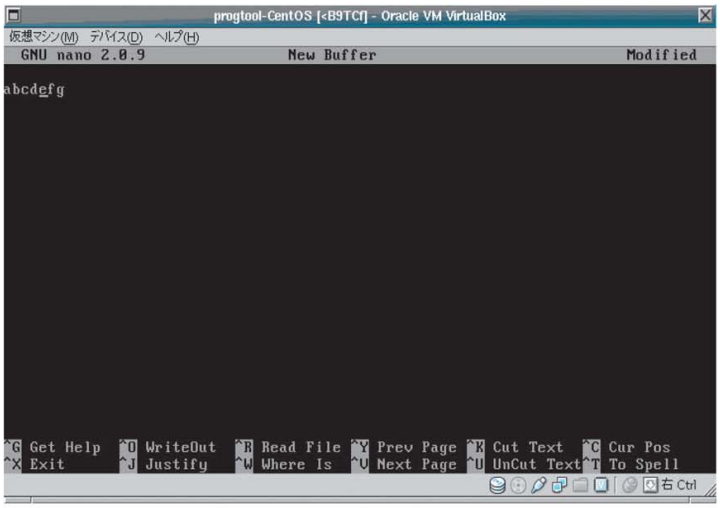


図4.9:「Ctrl」＋「X」でヘルプから抜ける

4.1.3 文章を編集してみよう

画面最下のメニューには，「^K Cut Text」や「^U UnCut Text」といったものもあります．

つまり「Ctrl」＋「K」で行をカットして，「Ctrl」＋「U」でペーストできるようです．

試してみましょう．まずはテスト用に**図4.10**のようにして，「ABCDEFG」「0123456789」を入力して行を追加します．

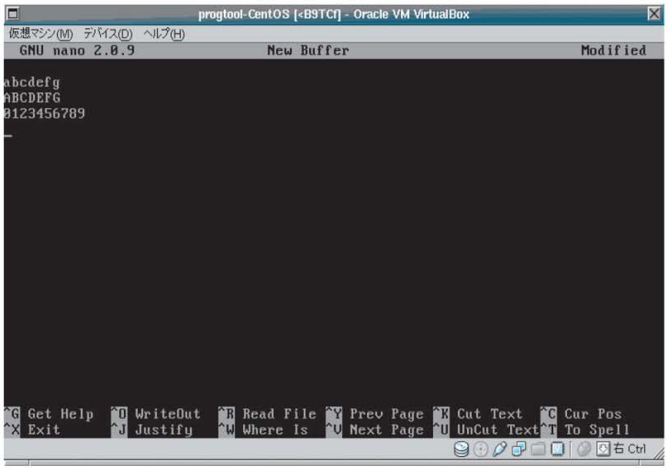


図4.10: 行を追加する

次に**図4.11**のように，カーソルを矢印キーで移動して「ABCDEFG」の行の位置に合わせます．

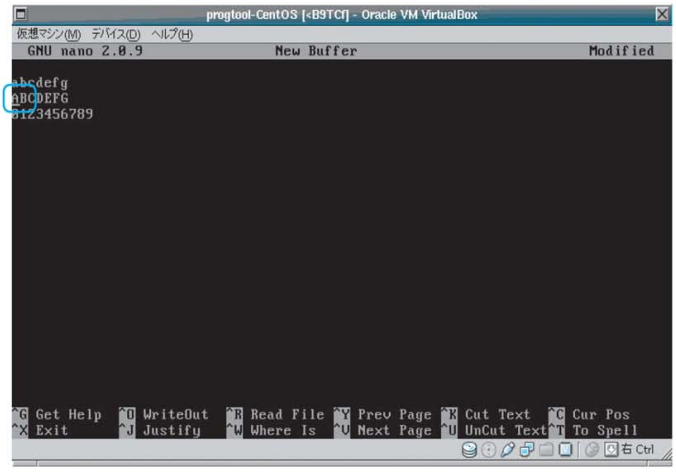


図4.11: カットする行にカーソルを移動する

ここでCtrlキーを押しながら「K」を押します．すると**図4.12**のように，「ABCDEFG」の行がカットされます．

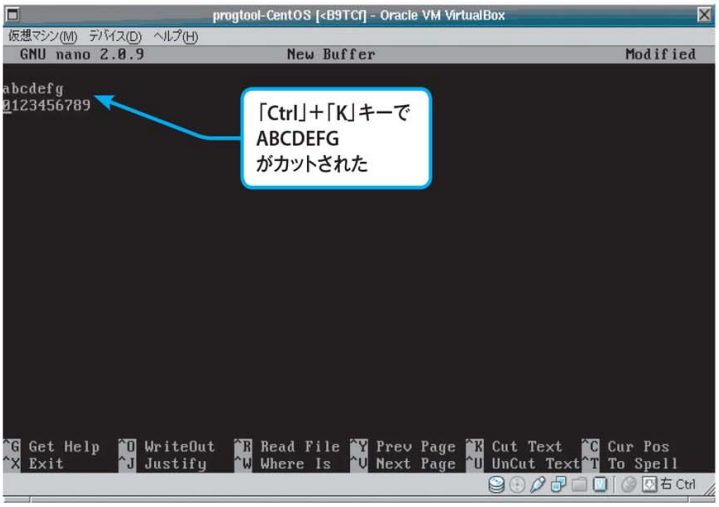


図4.12:「Ctrl」＋「K」で行をカットする

さらに矢印キーの「↑」を1回押して，カーソルを**図4.13**の位置に合わせます．

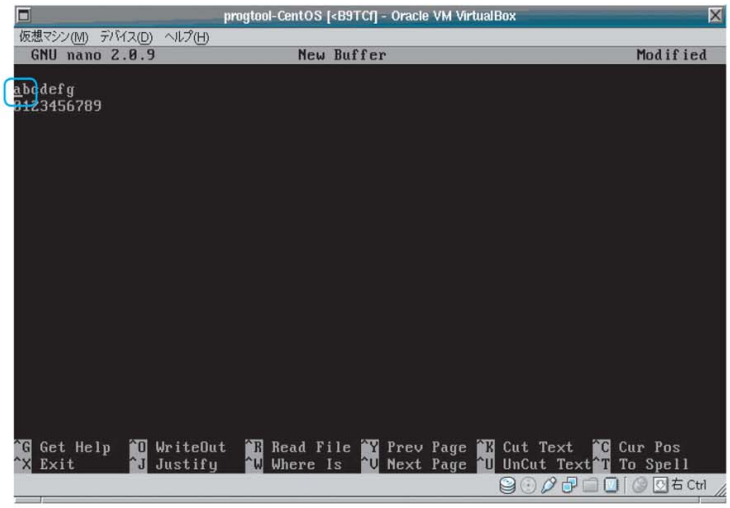


図4.13: ペースト位置にカーソルを移動する

ここでCtrlキーを押しながら「U」を押します．すると先ほどカットした「ABCDEFG」の行が，**図4.14**のようにカーソルのある行の位置に挿入されます．

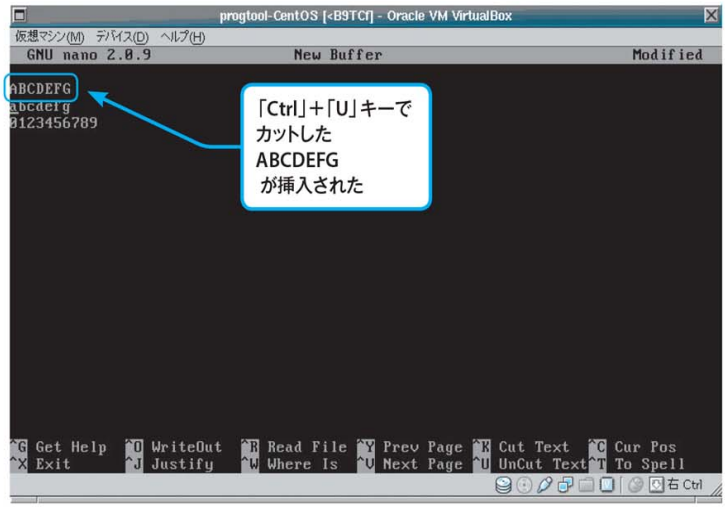


図4.14:「Ctrl」＋「U」で行をペーストする

4.1.4 文章を保存して終了しよう

画面の下方のメニューには，「^X Exit」の表示もあります．

つまりCtrlキーとXキーを押すことで，nanoを終了させることができるようです．

実際に「Ctrl」＋「X」を押してみると，**図4.15**のようになりました．

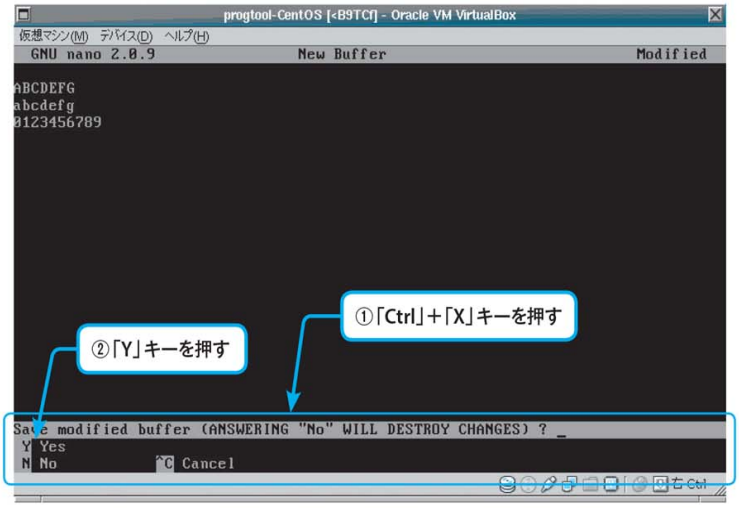


図4.15:「Ctrl」＋「X」を押して終了する

画面の下のほうにメッセージが表示され，「変更を保存するか？」と聞かれています．「Y」のキーを押してみましょう．

すると**図4.16**のように「File Name to Write:」として，保存のためのファイル名の入力を促されます．

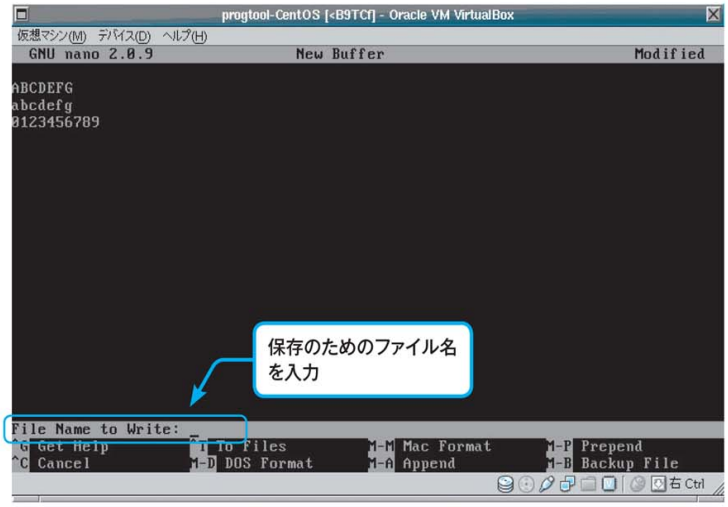


図4.16: ファイル名の入力

ここで「sample.txt」などのようにしてファイル名を入力し，Enterを押すと**図4.17**のように終了します．

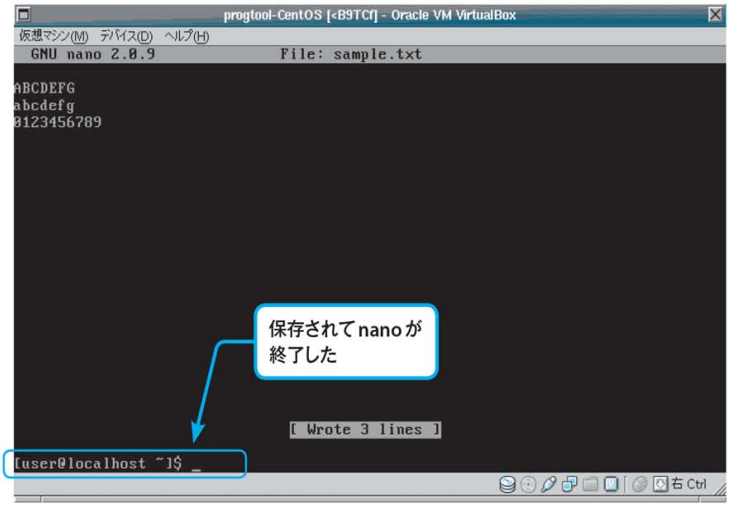


図4.17: ファイル名を入力すると，終了する

作成された「sample.txt」の内容を確認してみましょう．「cat sample.txt」を実行すると**図4.18**のようになり，確かにnanoで入力した内容がsample.txtに格納されています．

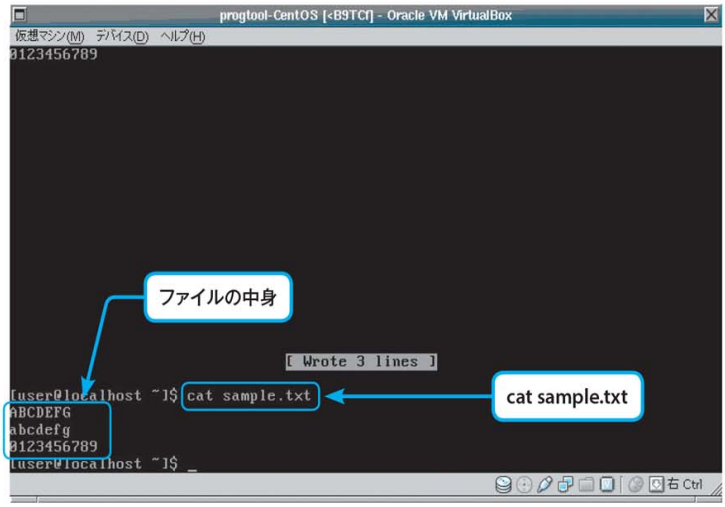


図4.18: sample.txtの内容を確認する

4.1.5 既存のファイルを編集しよう

既存のファイルを編集するには，以下のようにファイル名を引数にしてnanoを起動します．

```
[user@localhost ~]$ nano sample.txt
```

起動すると**図4.19**のようになりました．

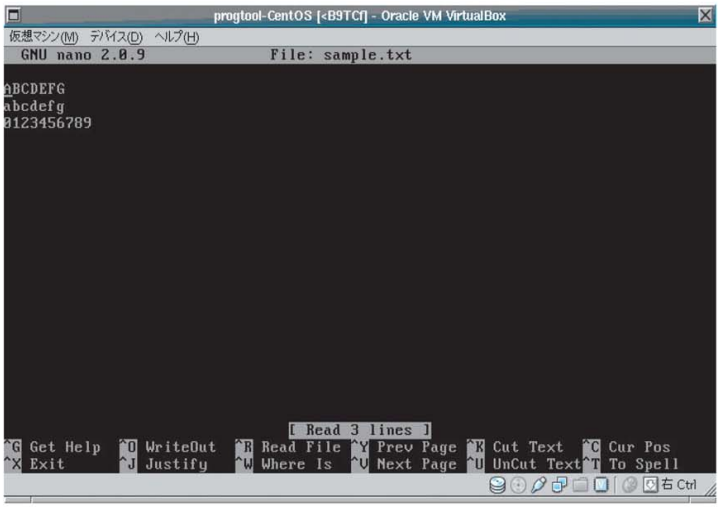


図4.19: nanoで「sample.txt」を開く

ここでファイルを適当に編集して，終了してみます．まずは「0123」と適当な文字を入力し，Enterで改行も入力します．

すると**図4.20**のようになります．

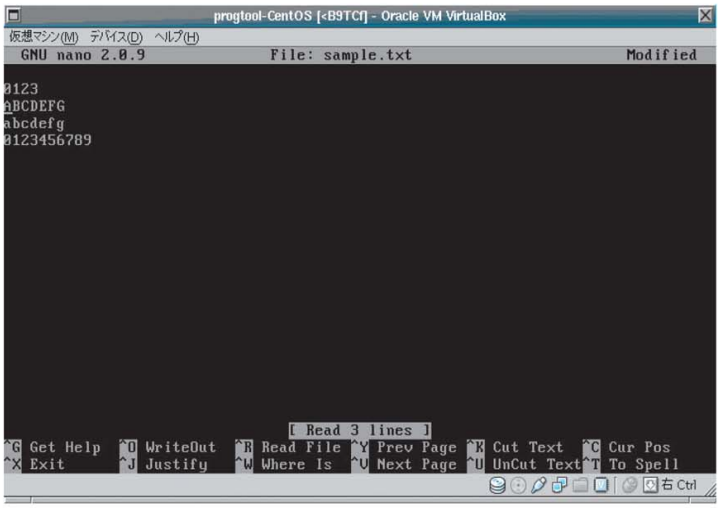


図4.20:「0123」の行を追加する

ファイルを編集したら，「Ctrl」＋「X」で終了しましょう．すると**図4.21**のようになり，修正内容を保存するかどうかを聞いてきます．

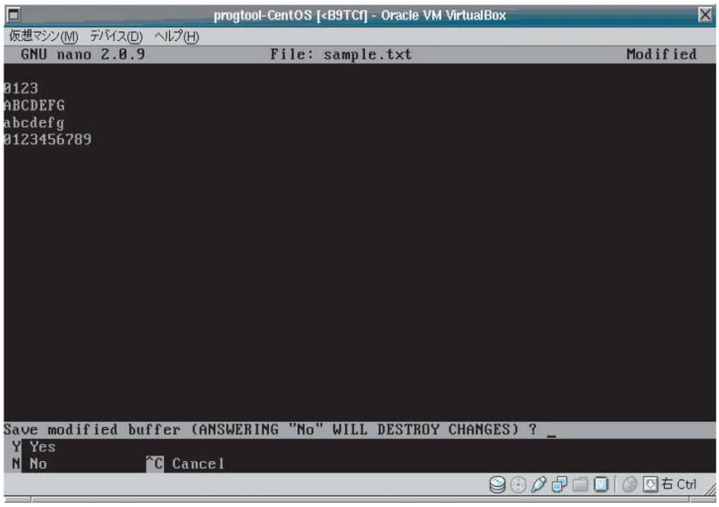


図4.21:「Ctrl」＋「X」で終了する

ここで「Y」を押すと，今度はファイル名の入力になります．

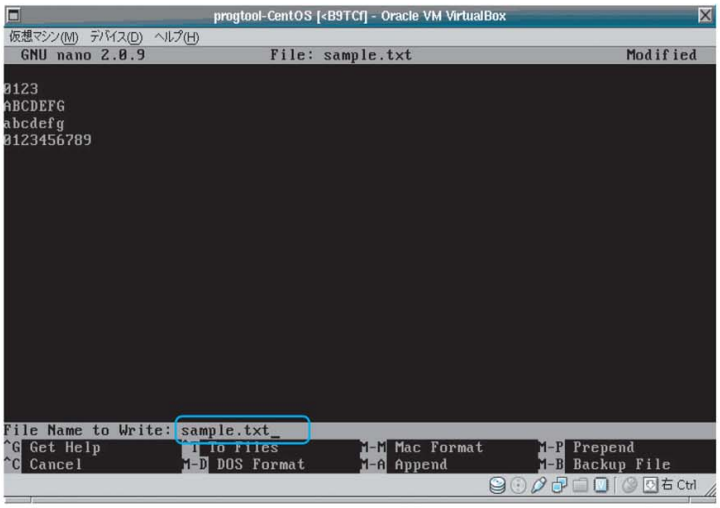


図4.22:「Y」を押すとファイル名の入力になる

今回はnanoをファイル名指定で起動しているため，**図4.22**ではファイル名として「sample.txt」が始めから指定されています．このままEnterを押せば，修正した内容を「sample.txt」として終了し，**図4.23**のようになります．

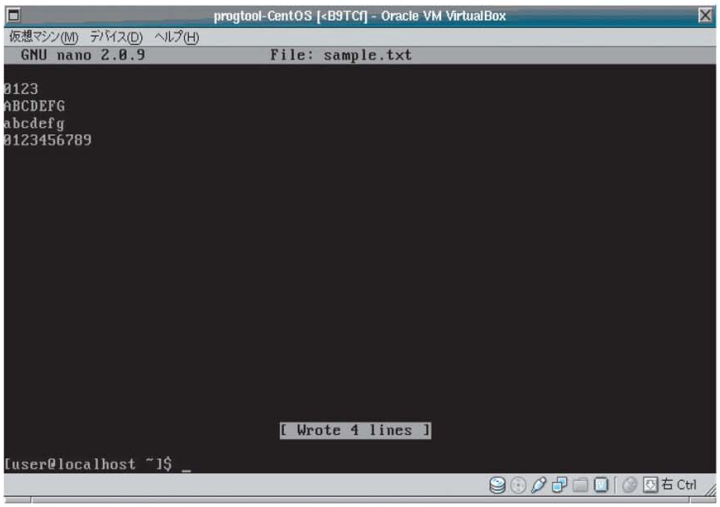


図4.23: nanoを終了する

終了したら，sample.txtの内容を確認してみましょう． catコマンドで sample.txt の内容を出力させてみると，**図4.24**のようになっています．先頭に「0123」という行が追加されていることがわかるでしょう．

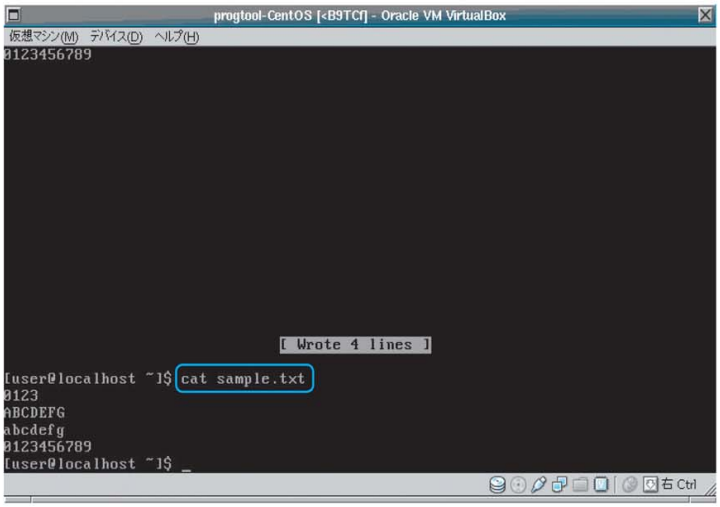


図4.24: 変更したsample.txtの内容を確認する

4.2 viエディタも使ってみよう

本書では簡単なプログラミングなどを行いますが，そのためにはテキストエディタを何かひとつ，使えるようにしておく必要があります．

そのためにnanoの使いかたを説明したわけですが，使いかたを知っておいたほうがいいテキストエディタが，実はもうひとつあります．

それは「vi（ヴィアイ）エディタ」というものです．

4.2.1 viエディタとは何か

viエディタは昔からあるテキストエディタです．ちょっと使いづらい点もあるのですが，慣れるとものすごく速くテキスト編集ができるため，人気のあるエディタでもあります．

なぜviエディタの使いかたを知っておいたほうがいいのかというと，以下のような理由があります．

viエディタは広く普及していて，標準的なテキストエディタと言えます．このためサーバなどでは，インストールされてないことはまず無いと言えます． CentOSにも標準で入っています．

例えばnanoを使うためには，そもそもnanoをインストールする必要があります．しかしインストールするには，ダウンロードのためにネットワークの設定をしなければならないかもしれません．

そしてそのためにはネットワーク周りの設定ファイルを編集しなければならないかもしれず，そのためにはテキストエディタが必要…ということになってしまうわけです．こういうのを「箱の鍵は箱の中状態」と言ったりします．

viエディタはたいていのシステムで，最初から入っていて使うことができます．このためviエディタの操作方法さえ知っていれば，このようなときにもとりあえず設定ファイルを編集することができます．

なので，最低限の使いかただけはマスターしておいたほうがいいエディタなわけです．

4.2.2 viエディタを起動してみよう

実際にviエディタを起動してみましょう．

```
[user@localhost ~]$ vi sample.txt
```

起動すると図4.25のような画面になったかと思います．先ほど作成したsample.txtの内容が表示されています．左上のアンダーバーは，カーソルのようです．

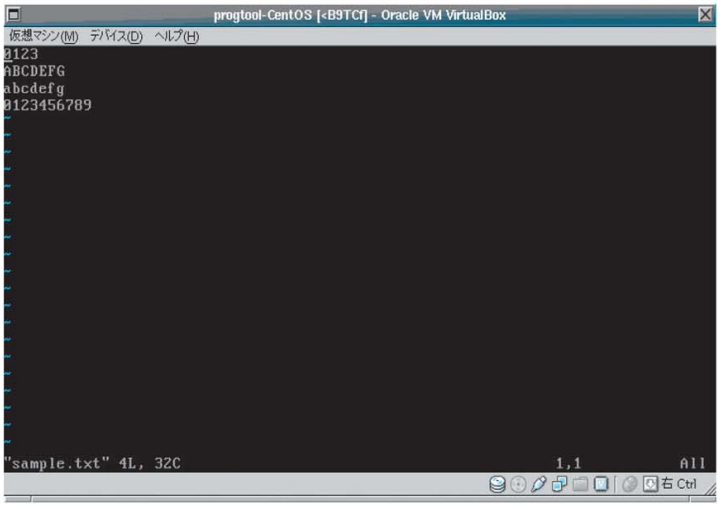


図4.25: viエディタを起動する

この状態で，矢印キーを適当に押してみましょう．押したキーに合わせて，図4.26のようにカーソルが移動します．

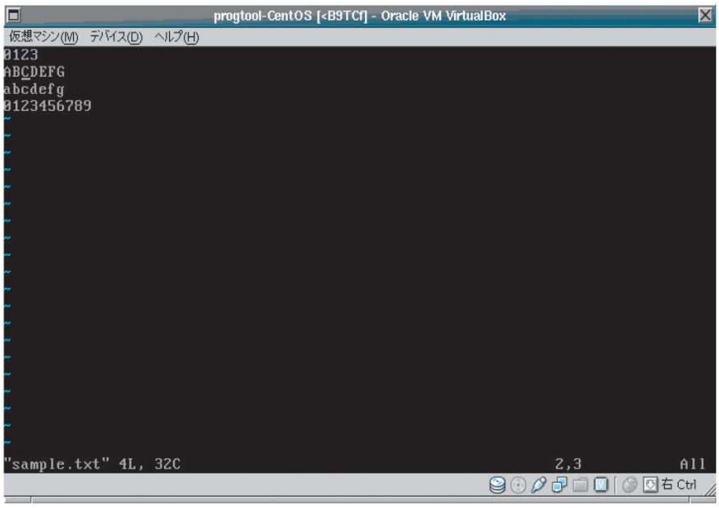


図4.26:「→」「→」「↓」のようにキーを押し，カーソルを移動させた

4.2.3 文章を入力してみよう

次にviエディタでの，文章の入力方法を説明します．

普通のテキストエディタは何も考えずにキーを押せばそれが入力されるのですが，viエディタはちょっと違います．

viエディタには「モード」という概念があり，「入力モード」と「コマンドモード」があります．文章の入力は「入力モード」で行いますが，起動時には「コマンドモード」になっています．

このため，このままいきなり文章を入力することはできません．

入力モードに入るには「I」のキーを押します．「I」はアルファベットの「アイ」です．「エル」や「1」ではないので注意してください．実際に「I」を押してみると，**図4.27**のような画面になります．

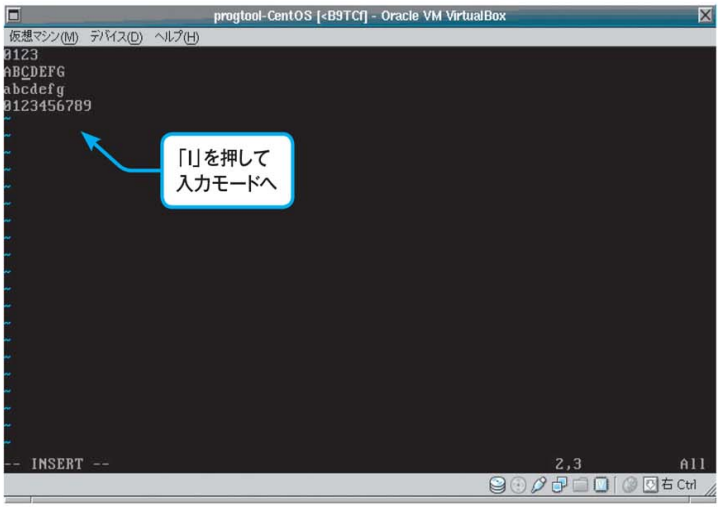


図4.27:「I」を押して入力モードに入る

画面下には「INSERT」と表示されており，入力モードに入っているようです．

これでキーボードから入力した文字が，カーソル位置に挿入されるようになります．ここで「A」「B」「C」とキーを押してみると，**図4.28**のようになりました．

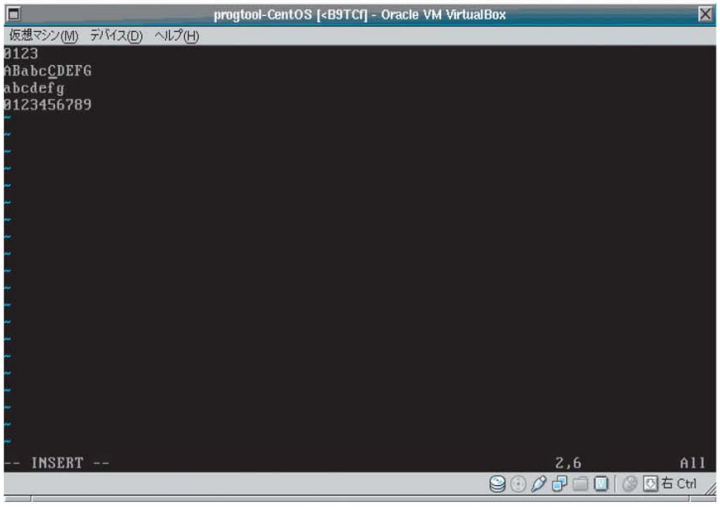


図4.28:「A」「B」「C」とキーを押す

カーソル位置に「abc」の３文字が挿入されています．

この状態で，矢印キーを押してカーソルを移動することもできます．「↓」のキーを２回押してみましょう．**図4.29**の位置にカーソルが移動しました．

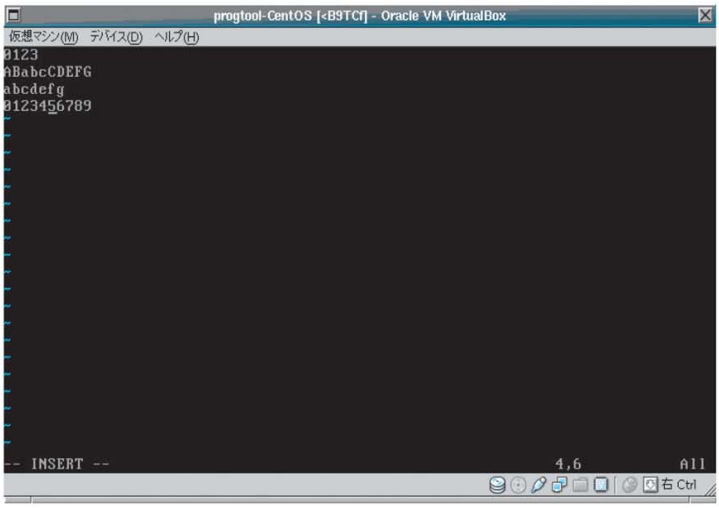


図4.29:「↓」のキーを２回押す

さらに「D」「E」「F」とキーを押してみます.

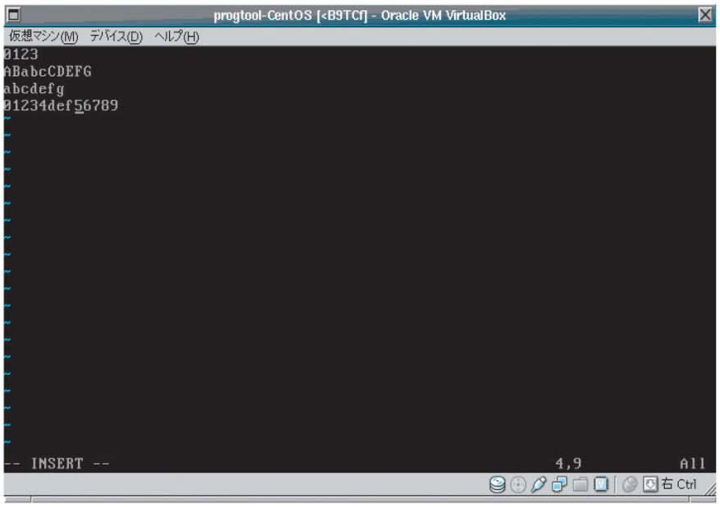


図4.30:「D」「E」「F」とキーを押す

図4.30のように，やはりカーソル位置に「def」の３文字が挿入されました.

4.2.4 文章を編集してみよう

次に，簡単な文章の編集です．

そのためには「入力モード」から「コマンドモード」に戻る必要があります．キーボードの左上にある「Esc」のキーを押すことで，入力モードからコマンドモードに戻ります．

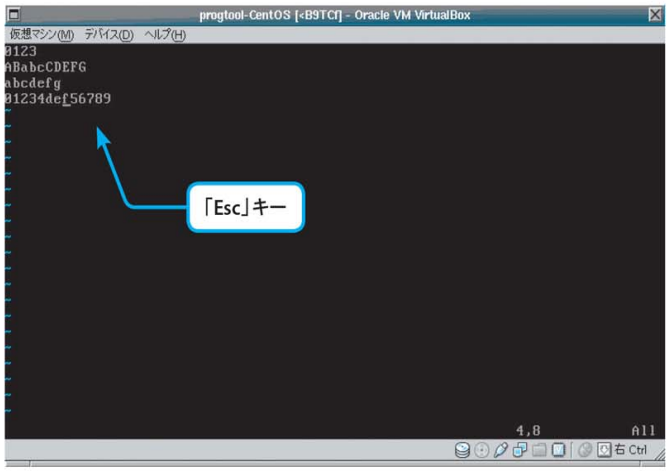


図4.31: Escキーでコマンドモードに戻る

図4.31では画面左下の「INSERT」の文字が消えていて，コマンドモードに戻ったことがわかります．

ここで「X」を押すことで，カーソルの位置の文字を1文字，削除します．やってみましょう．

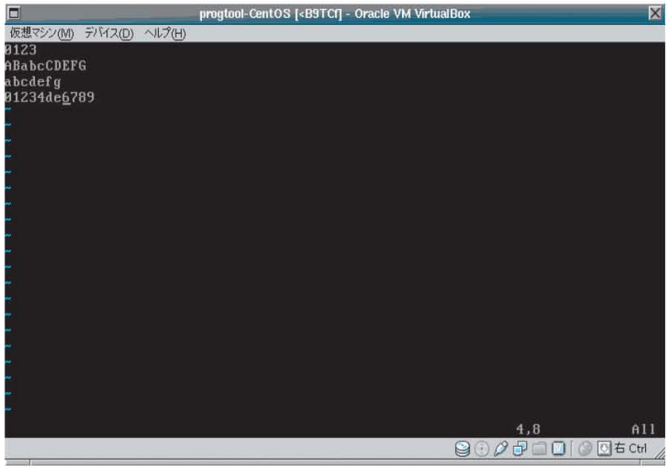


図4.32:「X」で文字を削除する

「X」を一回押すとカーソル位置にあった「ｆ」が削除され，もう一度押すと続けて「５」が削除されました．

行の移動は，「D」「D」のようにＤを２回押し，カーソルを移動させて「P」を押すことで可能です．

やってみましょう．まずは**図4.32**のようにカーソルが最下行の位置にある状態で，「D」「D」と続けてキーを押してみます．

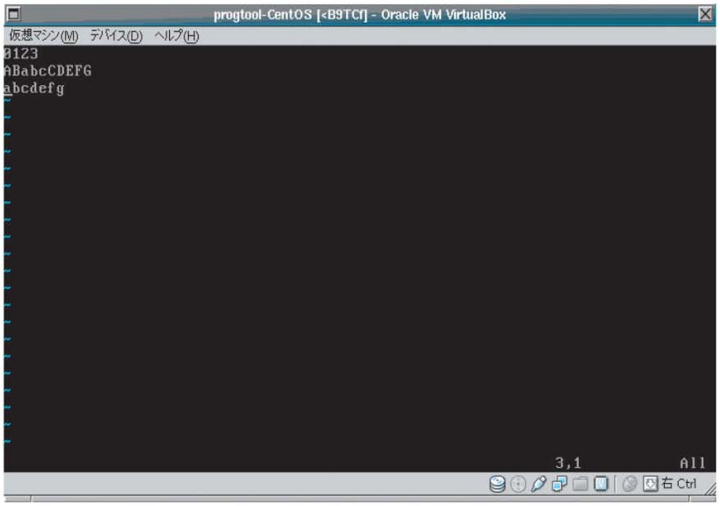


図4.33:「D」「D」で行を削除する

図4.33のように，最下行が削除されました．続けて「↑」を押してカーソルをひとつ上の行に移動します．

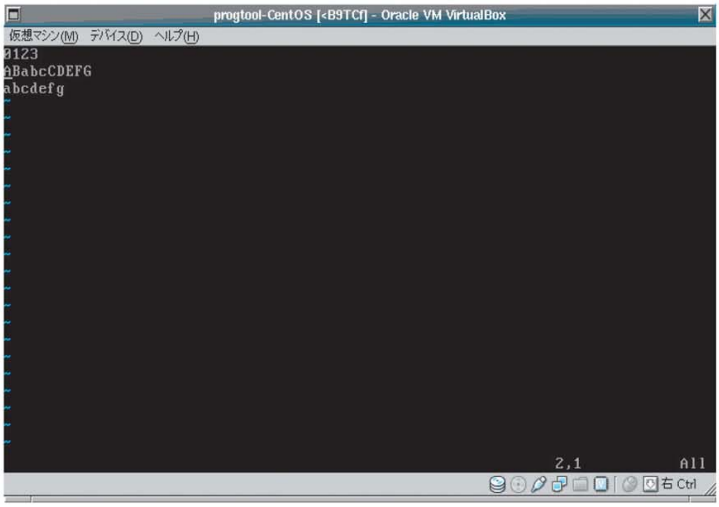


図4.34: ペーストする行にカーソルを移動する

図4.34のように，カーソルが2行目に移動しました．次に「P」を押すことで，先ほど削除した行をペーストします．

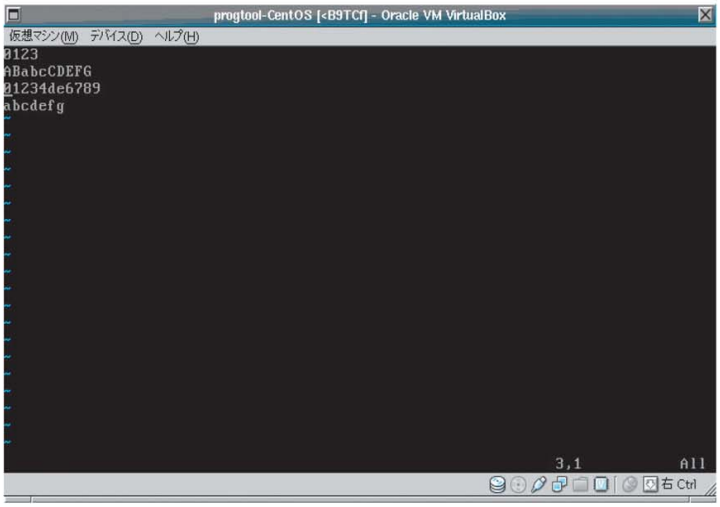


図4.35:「P」で行をペーストする

図4.35のようになり，最下行にあった「01234de6789」の行が，ひとつ上に移動できました．

現在はコマンドモードに戻っている状態ですが，さらに文章を入力したい場合には，「I」を押すことで入力モードに再び入ることができます．

4.2.5 文章を保存して終了しよう

最後に編集した文章の保存の方法を説明しましょう．

まず入力モードになっている場合には，Escキーを押してコマンドモードに戻ります．

さらに「:」を押してみましょう．これはShiftキーは押さずに，「:」のキーをそのまま押します．

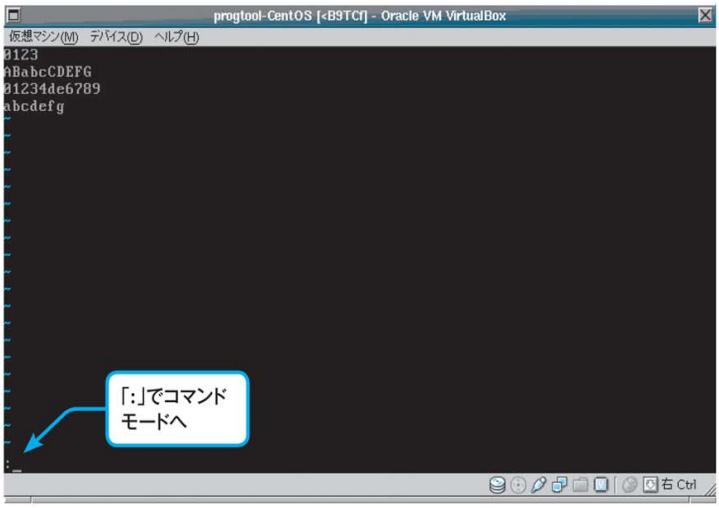


図4.36:「:」でコマンド入力に入る

図4.36のように画面左下に「:」が表示され，コマンド入力待ちになっています．

ここで「W」「Q」と押してみます．

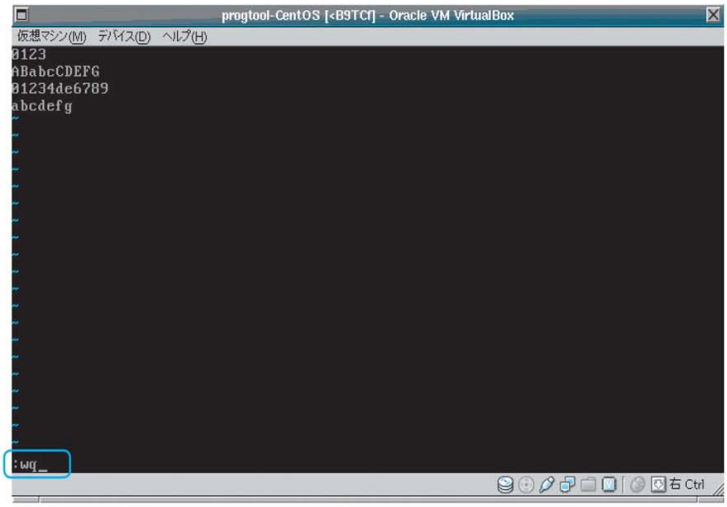


図4.37:「W」「Q」と押す

画面左下には**図4.37**のように，「wq」と表示されています．「w」はファイル書き込み，「q」は終了の意味です．

これに続けてEnterを押すことで，保存して終了します．

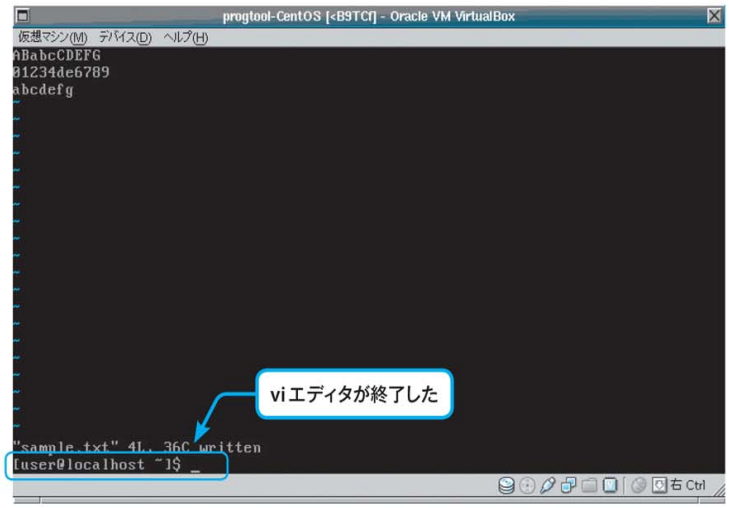


図4.38: Enterでファイルを保存して終了する

終了後にsample.txtの内容を確認すると，**図4.39**のようになりました．編集した内容を，ファイルに保存できているようです．

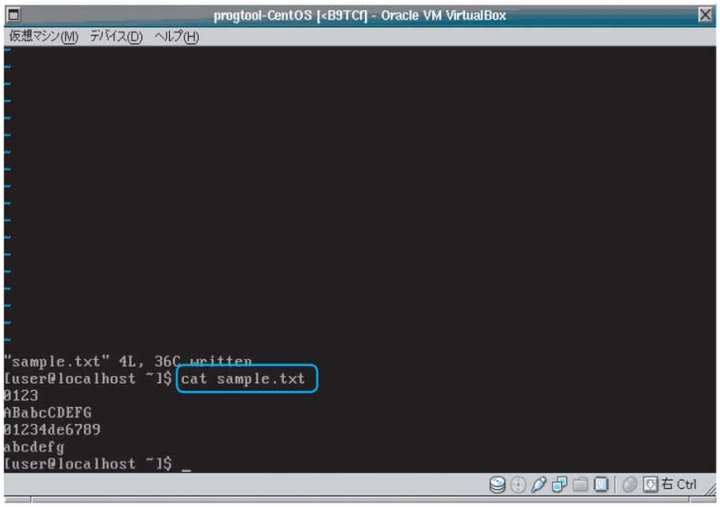


図4.39: sample.txtの内容を確認する

4.3 まとめ

viエディタは他にも機能が山のようにあるのですが，本書で説明するのはこれだけです．ちょっとした設定ファイルの編集ならば，これだけおぼえておけばだいたいなんとかなるでしょう．

テキストエディタは本書の内容を進めていく上で，非常に重要なツールになります．nanoはシンプルで使いかたも簡単です．またメニューも画面最下に常に表示されているため，迷うことは少ないでしょう．ぜひこの機会に，使いかたに慣れてみてください．

なおここでは，日本語の入力についてはとくに触れませんでした．もちろん実際には英数字だけではなく日本語の文章を読み書きしたい場合もあるわけですが，本書ではとりあえずは日本語入力のことは考えないことにします．

5

C言語に入門しよう

[C言語プログラミング編 1]ー文字列の出力

ここまででシェルによるファイル操作とテキストエディタによるファイル編集を体験し，プログラムを書くためのお膳立てはできました．

本章では，いよいよ「C言語でのプログラミング」に入門してみましょう．

プログラムはコンピュータに対する命令です．プログラムを書けるようになることで，コンピュータの使いかたの幅は広がります．世界が広がると言っても過言ではないでしょう．

とは言っても最初に扱うプログラムは簡単なものです．ぜひ恐れずに，プログラミングという新世界にチャレンジしてみてください．

5.1 簡単なプログラムを書いてみよう

最初に書くのは，「メッセージを出力するプログラム」です．まずは「hello.c」というファイルを作成するために，nanoを起動しましょう．

```
[user@localhost ~]$ nano hello.c
```

起動できたら，**リスト5.1**のような内容をファイルに書き込んでみましょう．

リスト5.1: hello.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(void)
005: {
006:         printf("Hello World!¥n");
007:         exit(0);
008: }
```

hello.cを作成する際の注意点は，「一字一句間違えずに入力する」ということに尽きます．ここで1字でも誤字があると，まず正常に動作しません．

もっとも間違えたところで，多くの場合はエラー出力されるので調べる手段はあるため，過剰に恐れることはありません．**図5.1**はnanoでhello.cを作成したところです．

これはC言語によるプログラムを記したファイルで，「ソースコード」と呼ばれるものです．「ソースファイル」や単に「ソース」と呼ばれることもあります．

C言語のソースコードのファイル名は，「XXX.c」というように拡張子を「.c」とすることが通例になっています．

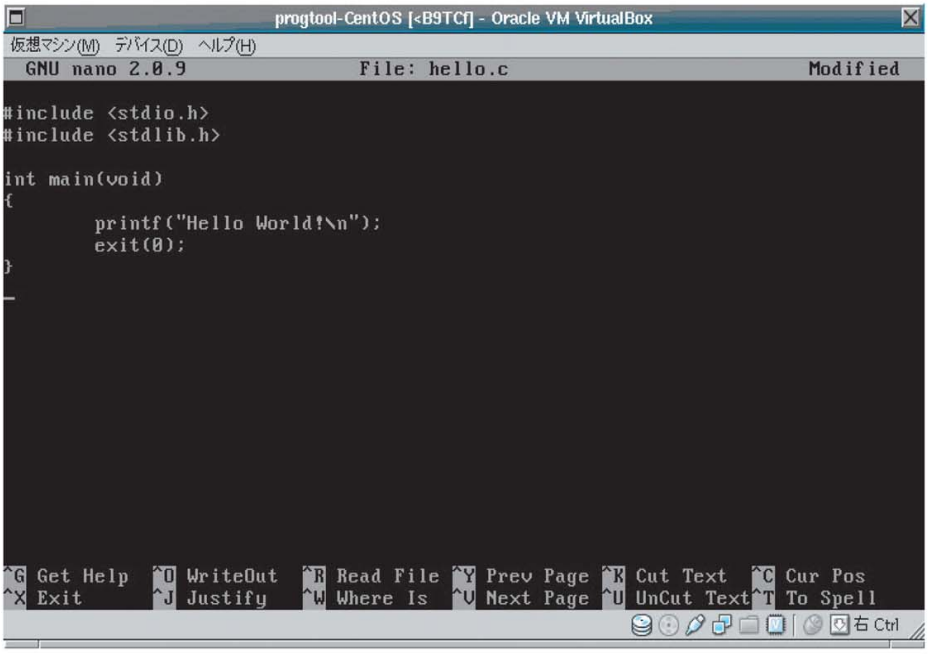


図5.1: hello.cを作成する

なお本書のＶＭ環境では日本語入力ができないため心配する必要は無いのですが，もしもWindowsなどの環境で hello.c を作成してCentOS側に持ってくるような場合には，「全角の空白が無いか」に注意してください．

また「"」や「;」などの文字も，全角文字になっていないか注意してください．これらは半角文字である必要があり，コンパイル時にエラーになってしまいます．とくに全角の空白が紛れ込んでいると気づきにくいため，要注意です．

5.1.1 gccでコンパイルしてみよう

hello.cが作成できたら，以下のようにコマンドを実行してみましょう．

```
[user@localhost ~]$ gcc hello.c -o hello
```

これは1章で説明した「GCC」という「コンパイラ」を使って，「コンパイル」という作業をしています．

ここでもしかしたら，エラーメッセージが出力されるかもしれません．これは「コンパイルエラー」と言われるものです．コンパイラであるgccコマンドが，hello.cの文法エラーを検出してコンパイルを中止している，という状態です．

例えば6行目の行末のセミコロン（「;」）が抜けていた場合には，以下のようなエラーが出力されます．このように1文字の不足でもエラーとなってしまうため，注意が必要なのです．

```
[user@localhost ~]$ gcc hello.c -o hello
hello.c: In function main:
hello.c:7: error: expected ; before exit
[user@localhost ~]$
```

エラーメッセージが出た場合，まずはその内容をきちんと読みましょう．

上の例で見ると，「exitの前に『;』を期待している」と言っています．「hello.c:7:」のように言われているのは「hello.cの7行目」という意味です．

どうやら7行目付近にセミコロンが無いようだ…という視点で見れば，直前の行である6行目の行末にセミコロンが無い，ということに気づくことができます．

この場合，hello.cをnanoで修正・保存して，再度コンパイルを行うことでエラーを消すことができます．

```
[user@localhost ~]$ nano hello.c          （6行目の行末にセミコロンを追加し…）
[user@localhost ~]$ gcc hello.c -o hello  （再度コンパイルを行う）
[user@localhost ~]$                       （今度はエラー無しで完了した）
```

5.1.2 コンパイルエラーに対処しよう

本書を実際に手を動かして試しながら読み進めている読者のかたの多くは，ここでおそらく何らかのコンパイルエラーが出ていることと思います．「一字一句間違えずに」と言われても，何かしらの間違いはしてしまうものでしょう．

ということで，ここでコンパイルエラーが出た場合の対処のコツを説明しておきましょう．

まずはエラーメッセージの読み方です．

- **エラーメッセージは先頭から見る**
  ある行にエラーがあると，後続の行も正常に解釈できず連鎖的にエラーになってしまうことがあります．この場合，後続のエラーが出ている行には間違いは無く，先頭のエラーを修正するときれいにエラーが消えることになります．エラーメッセージが複数出た場合には，まずは先頭のものから対処していき，修正するたびに再度コンパイルをしなおしてみましょう．
- **エラーが出ている行の，直前の行も見る**
  先述したセミコロン抜けの例がまさにそうなのですが，ある行に間違いがあった場合に，直後の行がうまく解釈できずにエラーとなることがあります．エラーメッセージが出力されている行だけでなく，その直前の行もチェックするようにしましょう．

また，もしもエラーになったら，以下のようなことを疑ってみてください．初心者がミスしそうなポイントをピックアップしてみました．

- **行末のセミコロン（;）が抜けていたりしないか？**
  先述したように，「error: expected ';' before '...'」というエラーになるようです．
- **「O」（大文字のオー）と「0」（数字のゼロ），「l」（小文字のエル）と「1」（数字のー）を間違えていないか？**
  2行目の「stdlib」は，小文字のエルです．また7行目の「exit(0)」のカッコの中は，数字のゼロです．初心者のうちはタイピングするのに精いっぱいで，単語の意味でなく文字のみを見てしまいがちなため，このようなミスが多く出るようです．
- **「main」を「nain」などとタイプミスしていないか？**
  「: undefined reference to `main'」というエラーになるようです．
- **全角文字になっている箇所が無いか？（すべて半角文字で入力します）**
  例えば先頭の「#include <...>」の部分の「<>」が全角文字になっていると，「error: #include expects "FILENAME" or <FILENAME>」というエラーになるようです．
- **全角の空白が入っていないか？**
  「error: stray '¥241' in program」というエラーになるようです．
- **必要な空白が抜けていないか？**
  例えば4行目の「int main()」の「int」と「main()」の間の空白は必要です．この空白が抜けていると，「: undefined reference to `main'」というエラーになります．

5.1.3 実行ファイルを実行してみよう

コンパイルエラーが出た箇所をひとつひとつ潰していって，「gcc hello.c -o hello」を実行してエラーが出力されなくなったら，コンパイルは成功しています．

生成されたファイルを，lsで確認してみましょう．

```
[user@localhost ~]$ ls hello*
hello  hello.c
[user@localhost ~]$
```

つまり「hello」というファイルが生成されているわけです．これは「実行形式ファイル」「実行ファイル」「実行形式」などと呼ばれるものです．

生成した「実行ファイル」は，次のようにして実行することができます．

```
[user@localhost ~]$ ./hello
Hello World!
[user@localhost ~]$
```

「Hello World!」と出力されれば，ひとまずプログラムの実行は成功です．

なお実行時には「./」のようにカレントディレクトリを指定しています．カレントディレクトリにあるファイルを実行する場合には，このように「./」を付加する必要があります．

5.1.4 コンパイルとは何か?

さて，これで「プログラムを入力してコンパイルし実行する」という一連の作業は完了したわけですが，途中で実行している以下のコマンドの意味は何なのでしょうか.

```
[user@localhost ~]$ gcc hello.c -o hello
```

先述したように，これは「コンパイル」と呼ばれる作業です.

「コンパイル」「コンパイラ」については1章で説明しましたが，実際にコンパイル作業を試してみて実感をつかんだところで，ここでもう一度おさらいしておきましょう.

コンピュータは**リスト5.1**のようなプログラムをそのまま実行できるわけではありません．コンピュータが実行できるのは，単なる数値の列として記述された「機械語」と呼ばれる言語だけです.

「コンピュータがそのまま実行できる」ということは，「そのようなプログラムを解釈して実行するための電子回路を持っている」という意味です.

しかし**リスト5.1**のようなプログラムを直接解釈するような電子回路は複雑すぎて，現実的ではありません．そこで「int」「main」のような単語でなく，数値をベースにして電子回路で解釈しやすい文法で設計されているコンピュータ向けの言語が「機械語」です.

このため我々が「機械語」を使ってプログラミングすればそれはそれで実行可能なのですが，それでは我々人間がプログラミングしにくい，といった問題があります.

よって，人間向けの「C言語」という言語でプログラムを書き，「gcc」という変換ツールを使えばC言語のプログラムを機械語に変換できるようになっています. このような作業を「コンパイル」，変換ツールのことを「コンパイラ」と呼びます.

我々は「C言語」という人間向けの言語でプログラミングを行えば，あとはコンパイラがうまく機械語に変換してくれるわけです.

5.2 プログラムを改造してみよう

出力するメッセージが「Hello World!」だけではなんだか面白みがありません．ということで，何か改造することを考えてみましょう．

リスト5.1ではいきなりC言語のプログラムを書き始めたわけですが，C言語の文法についてはまだ何も説明していません．

しかし，だからといって「プログラムの意味がまったくわからないので，改造などできない」なんていうことはありません．

リスト5.1とその実行結果をよく見てみると，以下のようなことに気がつきます．

- **実行すると「Hello World!」という文字列が出力されている**
- **リスト5.1には，「printf("Hello World!¥n");」のような行がある**
- **どうやら「printf(...)」に書かれた内容が出力されているようだ**
- **「printf(...)」の内容を変更すれば，出力される内容を変えることができるのではないか？**

プログラミングを効率良く習得するコツは，文法ばかりを覚えようとするのではなく，こうした推測と実験を繰り返すことです．

5.2.1 自分の名前を出力してみよう

hello.cのprintf()中の「Hello World!」という部分を変更して，自分の名前を出力するように改造してみましょう．

nanoを使ってhello.cを**リスト5.2**のように修正してみます．修正したのは6行目の，printf()の部分です．

リスト5.2: hello.c（修正版）

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(void)
005: {
006:         printf("My name is SAKAI Hiroaki.¥n");
007:         exit(0);
008: }
```


リスト5.2では「SAKAI Hiroaki」のようにして筆者の名前を入れてありますが，この部分には，ぜひ読者のあなたの名前を入れてみてください．

改造ができたらファイルを保存し，コンパイルしなおして実行してみましょう．

```
[user@localhost ~]$ gcc hello.c -o hello
[user@localhost ~]$ ./hello
My name is SAKAI Hiroaki.
[user@localhost ~]$
```

名前が出力されれば成功です．

コンパイルをしなおすことで，実行ファイルの「hello」が再生成されます．

つまりhello.cを修正しただけでgccを実行するのを忘れると，出力文字列が「Hello World!」のままということになりますので注意してください．「hello.cを変更したのに表示内容が変化しない」というかたは，コンパイルしなおしているかどうかを確認してみてください．

5.2.2 複数の文字列を出力してみよう

出力文字列を変更することで，どうやら「printf()」というものが文字列を出力するための命令らしきものだということが漠然と想像できてきます．

では，printf()を複数記述したらどうなるのでしょうか？

プログラミングを習得するためには，このような疑問を持ったらまずは試してみることが大切です．

今まではhello.cというファイルを修正してきましたが，ここからは以前のファイルを残すために，別のファイルを作成して修正することにしましょう．

まずは改造用に，hello.c を multi.c という別ファイルにコピーします．

```
[user@localhost ~]$ cp hello.c multi.c
```

さらに nano を使うことで，multi.c を**リスト 5.3** のように修正してみます．

```
[user@localhost ~]$ nano multi.c
```

リスト5.3: multi.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(void)
005: {
006:         printf("Hello World!¥n");
007:         printf("My name is ...¥n");
008:         printf("SAKAI Hiroaki.¥n");
009:         exit(0);
010: }
```

リスト 5.3 では，printf() の行を新たに追加しています．

これをコンパイルして実行してみましょう．

```
[user@localhost ~]$ gcc multi.c -o multi
[user@localhost ~]$ ./multi
Hello World!
My name is ...
SAKAI Hiroaki.
[user@localhost ~]$
```

printf() によって記述された 3 つの文字列が，順に出力されています．ということは printf() の記述は，上から順に実行されていくようです．

5.2.3 改行コードを調べてみよう

ところでここまでで，printf() によって出力する文字列には，「"Hello World!¥n"」のようにして，「¥n」という謎の文字が入っています．これはいったい何のおまじないでしょうか？

疑問に思ったら試してみましょう．まず multi.c を newline.c というファイルにコピーします．

```
[user@localhost ~]$ cp multi.c newline.c
```

さらにnewline.cを，**リスト5.4**のように修正してみます．

```
[user@localhost ~]$ nano newline.c
```

リスト5.4: newline.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(void)
005: {
006:         printf("Hello World!¥n¥n¥n");
007:         printf("My name is ...");
008:         printf("SAKAI Hiroaki.¥n");
009:         exit(0);
010: }
```

newline.cでは３つあるprintf()のうち，１つ目には「¥n¥n¥n」のようにして「¥n」を余分に加え，２つ目からは「¥n」を削ってみました．

コンパイルして実行してみましょう．

```
[user@localhost ~]$ gcc newline.c -o newline
[user@localhost ~]$ ./newline
Hello World!


My name is ...SAKAI Hiroaki.
[user@localhost ~]$
```

１つ目の「printf()」による「Hello World!」の後には２つの空行が続いています．

また２つ目の「printf()」による「My name is ...」の後に，改行されずに名前が出力されるようになりました．

これは，どういうことでしょうか？

まずは想像してみましょう．プログラミングを習得するためには，このように試してみた結果から，理由を推測することが遠回りに見えて大きな近道になります．

これは，「¥n」が「改行」を表していると考えると納得ができます．

１つ目の「printf()」では「¥n¥n¥n」のように３つの改行が続いているため，２つの空行が追加されています．

２つ目の「printf()」では「¥n」が無いため，後続の名前の出力が改行されずにくっついてしまっているわけです．

5.2.4 順次実行されていることを確認しよう

ここまでで，printf() による文字列出力の動作が漠然とわかってきました．まとめると，以下のようになっているようです．

- **main() { ... } の中に記述された部分が，上から順に実行される**
- **printf("...") と書くと，"..." の部分が出力される**

「上から順に実行される」ということを，もっとわかりやすい形で確認できないでしょうか？

例えば newline.c を sleep.c というファイルにコピーして，sleep.c では**リスト 5.5** のように printf() の間に「sleep(1)」という行を追加してみましょう．

```
[user@localhost ~]$ cp newline.c sleep.c
[user@localhost ~]$ nano sleep.c
```

リスト 5.5: sleep.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003: #include <unistd.h>
004:
005: int main(void)
006: {
007:         printf("Hello World!¥n");
008:         sleep(1);
009:         printf("My name is ...¥n");
010:         sleep(1);
011:         printf("SAKAI Hiroaki.¥n");
012:         sleep(1);
013:         exit(0);
014: }
```

sleep.c をコンパイルして実行すると，どうなるでしょうか．

```
[user@localhost ~]$ gcc sleep.c -o sleep
[user@localhost ~]$ ./sleep
Hello World!
```

実行すると，まず「Hello World!」と出力されます．

さらに 1 秒後には，以下のようになります．

```
[user@localhost ~]$ gcc sleep.c -o sleep
[user@localhost ~]$ ./sleep
Hello World!
My name is ...
```

このように，続けて「My name is ...」が出力されます．さらに 1 秒後には，以下のようになります．

```
[user@localhost ~]$ gcc sleep.c -o sleep
[user@localhost ~]$ ./sleep
Hello World!
My name is ...
SAKAI Hiroaki.
```

名前が出力されています．

そして最後に 1 秒待つと，プログラムは終了します．

```
[user@localhost ~]$ gcc sleep.c -o sleep
[user@localhost ~]$ ./sleep
Hello World!
My name is ...
SAKAI Hiroaki.
[user@localhost ~]$
```

sleep(1) は「1 秒待つ」という意味です．printf() による文字列出力の合間に 1 秒の待ち合わせを行い，逐次処理を進めているわけです．

このことからも「main()」の内部に書いた行が，上から順に実行されているということがわかります．

5.2.5 exit(0) の動作を調べてみよう

ところで main() の末尾には，「exit(0)」という行があります．これはプログラムを終了するという意味になります．

ということは，exit() の後に printf() を書くとどのような動作になるのでしょうか？

試してみましょう．sleep.c を after.c というファイルにコピーして，**リスト 5.6** のように修正します．

```
[user@localhost ~]$ cp sleep.c after.c
[user@localhost ~]$ nano after.c
```

リスト 5.6: after.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003: #include <unistd.h>
004:
005: int main(void)
006: {
007:         printf("Hello World!¥n");
008:         sleep(1);
009:         printf("My name is ...¥n");
010:         sleep(1);
011:         exit(0);
012:         printf("SAKAI Hiroaki.¥n");
013:         sleep(1);
014: }
```

コンパイルし実行した結果は，次のようになりました．

```
[user@localhost ~]$ gcc after.c -o after
[user@localhost ~]$ ./after
Hello World!
My name is ...
[user@localhost ~]$
```

名前が出力されなくなってしまいました．

ということは exit(0) の位置で，プログラムの実行は終了しているようです．

5.3 文字列出力で遊んでみよう

ここまではプログラムを様々に改造して動作を確認することで，理解を深めてきました．

しかしプログラミングを体験するときには「遊び」も大切です．ここでちょっと遊びの要素も入れてみましょう．

5.3.1 色を出してみよう

特定の文字列を出力することで，色を出したりといった特殊効果を入れることができます．これは「エスケープシーケンス」などと呼ばれます．

例えば**リスト5.7**のようなプログラムを試してみてください．ファイル名はcolor.cとします．

リスト5.7: color.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003: #include <unistd.h>
004:
005: int main(void)
006: {
007:         printf("Hello World!¥n");
008:         sleep(1);
009:         printf("¥033[31m");
010:         printf("This color is red.¥n");
011:         sleep(1);
012:         printf("¥033[0m");
013:         printf("This color is normal.¥n");
014:         sleep(1);
015:         exit(0);
016: }
```

実行すると次ページの**図5.2**のようになります．

```
[user@localhost ~]$ gcc color.c -o color
[user@localhost ~]$ ./color
Hello World!
This color is red.     (赤色で出力される)
This color is normal.  (通常色で出力される)
[user@localhost ~]$
```

「This color is red.」の部分は赤で出力されます．つまり"¥033[31m"という文字列を出力すると，その後は赤色での表示になります． "¥033[0m"を出力すると，通常の色に戻ります．

このような特殊文字列を出力することで，出力時の色を制御することができるわけです．

図 5.2 は実行時の画面です．雰囲気だけでも感じでいただけるでしょうか？

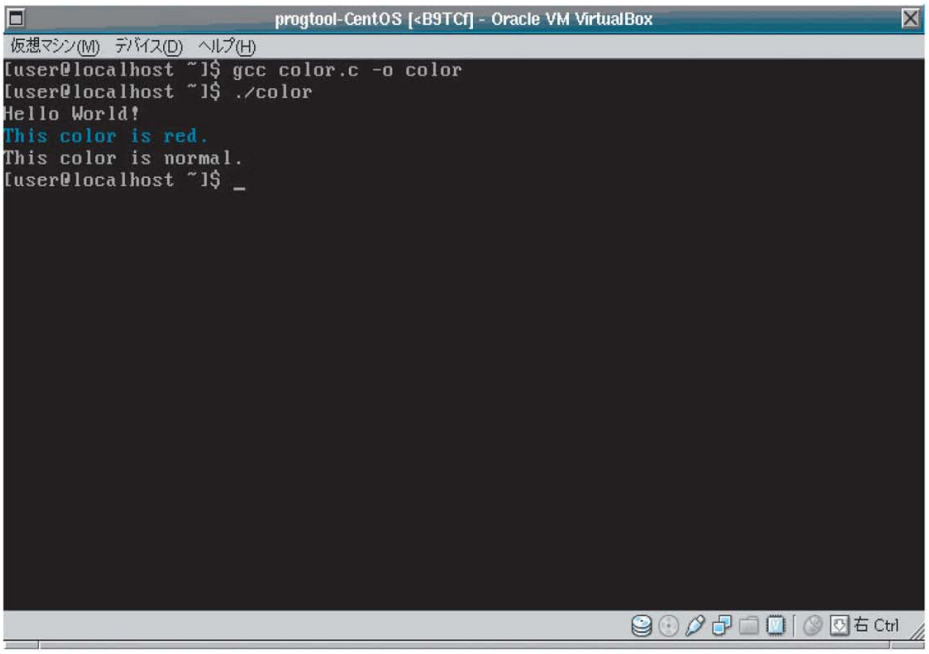


図5.2: 色を使う

5.3.2 アニメーションに挑戦しよう

最後にエスケープシーケンスを利用して，アニメーションに挑戦してみましょう．

アニメーションとはいっても画像によるグラフィカルなものではなく，テキスト文字を利用したシンプルなものです．しかしそのような単純なものでも，何かが動くということは楽しいものです．

まずは**リスト5.8**のような内容で，animation.c を作成してください．

リスト5.8: animation.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003: #include <unistd.h>
004:
005: int main(void)
006: {
007:         printf(" O/¥n");
008:         printf("/| ¥n");
009:         printf("<< ¥n");
010:         sleep(1);
011:         printf("¥033[3A");
012:         printf("<O/¥n");
013:         printf(" | ¥n");
014:         printf(" >>¥n");
015:         sleep(1);
016:         printf("¥033[3A");
017:         printf(" O ¥n");
018:         printf("/|>¥n");
019:         printf("/< ¥n");
020:         sleep(1);
021:         printf("¥033[3A");
022:         printf(" O/¥n");
023:         printf("<| ¥n");
024:         printf("/ >¥n");
025:         sleep(1);
026:         exit(0);
027: }
```

リスト5.8では "¥033[3A" のような制御文字列を出力しています．これは現在のカーソルを上に３つ，移動するというものです．

リスト5.8では人型のテキストを出力して１秒待った後，上に戻ってまた別の人型を表示しています．結果として人が動いているようなアニメーションになります．

実行してみると，以下のように人が踊り出します．

```
[user@localhost ~]$ gcc animation.c -o animation
[user@localhost ~]$ ./animation
 O/
/|
<<
```

```
[user@localhost ~]$ gcc animation.c -o animation
[user@localhost ~]$ ./animation
<O/
 |
 >>
```

```
[user@localhost ~]$ gcc animation.c -o animation
[user@localhost ~]$ ./animation
 O
/|>
/<
```

```
[user@localhost ~]$ gcc animation.c -o animation
[user@localhost ~]$ ./animation
 O/
<|
/ >
```

5.4 機械語を見てみよう

さて，コンピュータはC言語のプログラムを直接理解することはできず，「機械語」「マシン語」と呼ばれる言語に変換して実行されるということを説明しました．

objdumpというコマンドを利用することで，その「機械語」というものを直接見てみることもできます．

せっかくなので見てみましょう．ここで「機械語」が理解できる必要はありませんが，なんとなく見ておくことも，良い体験になるでしょう．

5.4.1 「逆アセンブル」をしてみよう

実行ファイル「hello」に対して以下を実行してみましょう．これは「逆アセンブル」と言われる操作になります．

```
[user@localhost ~]$ objdump -d hello > hello.dis
```

生成されたhello.disを見てみると，以下のような部分が見つかります． lessで表示させ，「main」という部分を検索してみてください． lessでの検索は，「/」を押すことで行えます．

```
[user@localhost ~]$ less hello.dis
...（中略）...
080483e4 <main>:
 80483e4:       55                      push   %ebp
 80483e5:       89 e5                   mov    %esp,%ebp
 80483e7:       83 e4 f0                and    $0xfffffff0,%esp
 80483ea:       83 ec 10                sub    $0x10,%esp
...（後略）...
```

これがプログラムの本体です．

5.4.2 機械語は，数字の羅列になっている

機械語について，ちょっとだけ説明しておきましょう．

中央付近にある「55」や「89 e5」といったものが，実は機械語と呼ばれるものになります．機械語に関しては本書の範囲ではないため，ここではまあそうなんだくらいの理解で構いません．

こうして見ると，コンピュータは「55」「89 e5」「83 e4 f0」のような命令を逐次実行していくわけです．このため我々人間が機械語で直接プログラミングをするとしたら，「55」「89 e5」「83 e4 f0」のようなものを書いていくことになります．

しかしそれではたいへん面倒であるため，C言語のような人間向けのプログラミング言語があり，コンパイラがそれを機械語に変換することで実行ファイルを作成しているわけです．

機械語コードの意味はわからなくとも，面倒そうだということはわかります．このように「わからなくても試しに見てみる」ということで，感覚的に感じてみることは非常に大切なことだと思います．

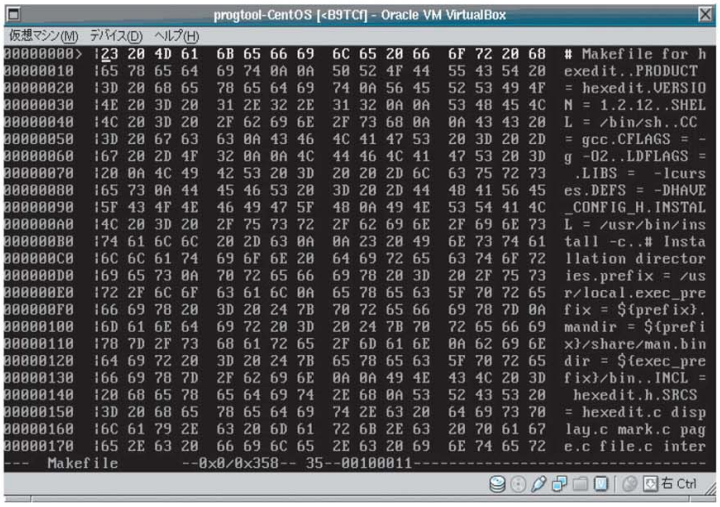


図5.3：バイナリエディタ hexeditで見たテキストファイル
（「14.4 フリーソフトウェアをコンパイルしてみよう」参照）

5.5 まとめ

プログラミングを習得する際にまず重要なことは,「慣れること」です．そのためには本書で説明したことのみ行うのではなく，いろいろに修正して試してみることが大切です．

本章ではプログラミングというものに入門したばかりです．しかしだからといって,「プログラムを改造するほどの知識は無い」と思う必要はありません．ここまでに説明した内容だけでも,「出力するメッセージを増やす」「出力するメッセージを変更する」といった改造は可能なはずです．

むしろ習得が早いひとほど，こうした簡単な「改造」を繰り返しているものです．ぜひプログラムをいろいろに改造して，プログラミングというものにまずは慣れてみてください．

●本書で紹介するCentOSのCUIコマンド－1（1・2章）

利用ユーザ名：user　**初期パスワード**：progtooluser
su ユーザ名：root　**初期パスワード**：progtoolroot

コマンド	ツールの役割	ページ
whoami	現在のユーザ名を表示するコマンド	31, 32
loadkeys jp106	日本語キーボードを利用する... [スーパーユーザで実行]	33
exit	ログアウト	34
passwd	パスワードの変更	35, 37
su	一般ユーザから「スーパーユーザ」に切り替える	39
su（一般ユーザ名）	スーパーユーザから一時的に「一般ユーザ」に切り替える	41
yum -y update	CentOSを最新版に更新... [スーパーユーザで実行]	44
init 0	シャットダウンで終了... [スーパーユーザで実行]	45
startx	GUIの画面にする	46
touch（ファイル名）	中身が空の（ファイル名）を作成する	51
ls	ファイル一覧を表示する。「list」の略	51
ls（ファイル名）	（ファイル名）のファイルがあることを確認する	52
ls -l	ファイル一覧をファイルの詳細情報付きで表示	52
↑	実行したコマンドの履歴を遡って順番に表示	60
cat（ファイル名）	（ファイル名）の内容を参照する	65
echo（文字列）	（文字列）を単に表示する	65
echo（文字列）>（ファイル名）	（文字列）を（ファイル名）への書き込む... [上書き]	65
echo（文字列）>>（ファイル名）	（文字列）を（ファイル名）への書き込む... [追記]	65
mkdir（ディレクトリ名）	（ディレクトリ名）のディレクトリを作成する「make directory」の略	67
cd（ディレクトリ名）	（ディレクトリ名）のディレクトリへ移動する「change directory」の略	68
cp（ファイル名1）（ファイル名2）	（ファイル名1）をコピーして（ファイル名2）を作成する	70
mv（ディレクトリ名）/（ファイル名）.	（ディレクトリ名）にある（ファイル名）を . 現在のディレクトリ［カレントディレクトリ］に移動する	70
*	［ワイルドカード］ すべての文字列	72
?	［ワイルドカード］ 任意の1文字	72
rm（ファイル名）	（ファイル名）を削除する。「remove」の略	74

6

複数のスクリーンを使おう
[ツール編2]—screenコマンド

ここまでCUIによる操作を体験してきて，「コマンド操作できる画面がひとつしか無くて不便」と思った読者のかたも多いのではないでしょうか．

確かにGUIベースの操作ならば，必要な数だけウィンドウを開いて作業することができます．しかしこれは，GUIだけのものではありません．CUIでも同様に，複数の「スクリーン」を切り替えて操作できるような枠組があります．

ここで説明するのは「screen」というコマンドです． CUIでの操作がベースになるとき，ぜひ覚えておいたほうがいいツールです．

6.1 screenコマンドで複数のスクリーンを開こう

screenコマンドを使うことで，複数の画面を切り替えながら操作をすることができるようになります．

つまり，こちらのスクリーンではテキストエディタを開いて，別のスクリーンではコンパイル作業，そして別のスクリーンではメールチェックといったように，画面を使い分けることができるわけです．

6.1.1 screenを起動する

まずは「screen」を起動してみましょう．シェル上から，おもむろに以下のコマンドを実行してみます．

```
[user@localhost ~]$ screen
```

すると，**図6.1**のような画面になるでしょう．

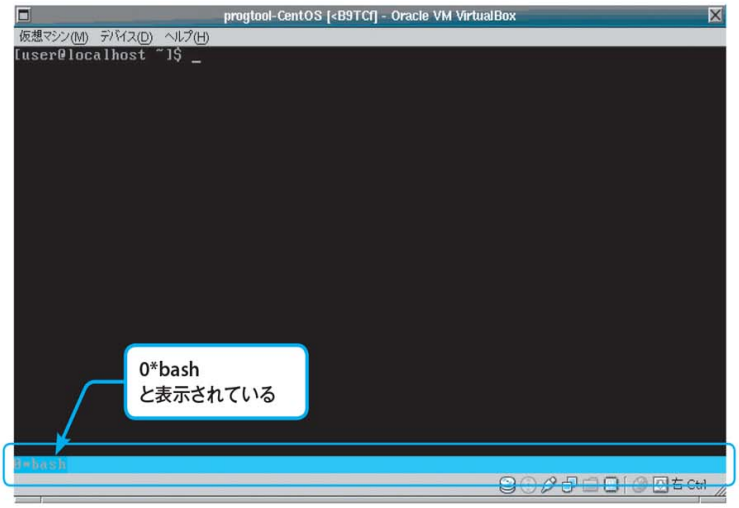


図6.1: screenを起動する

あまり変化は無いようですが，よく見ると画面最下部にバーが出ています．screenコマンドでは，ここに各スクリーンの情報が表示されます．

screenを実行したあとの操作は，今までと同じです．ためしにwhoamiを実行してみましょう．

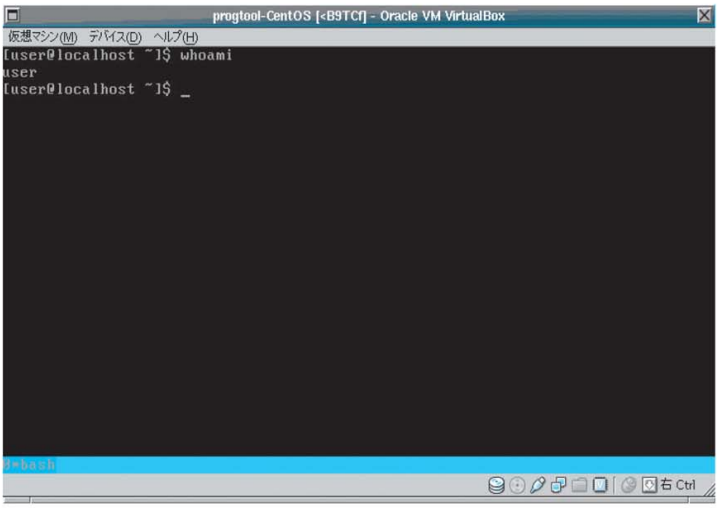


図6.2: whoamiを実行する

図6.2のように，問題なくwhoamiコマンドが実行できました．

シェルでのコマンド操作には，とくに変わりは無いようです．

6.1.2 新しいスクリーンを開く

では，別の新しいスクリーンを開いてみましょう．

screenコマンドには「エスケープキー」として定義されているキーがあります．本書のVM環境では「Ctrl」＋「J」に割り当てられています．「エスケープキー」というのは，何かの機能を実行するために最初に押す特殊なキー割り当てのことです．

新しいスクリーンは，エスケープキーに続けて「C」を押すことで開くことができます．
つまり本書のVM環境では，「Ctrl」＋「J」→「C」のようにキー入力すると，新しいスクリーンが開きます．「C」を押すときには，Ctrlキーは押したままでも離してしまっても構いません．

実際に押してみると，**図6.3**のような画面になります．

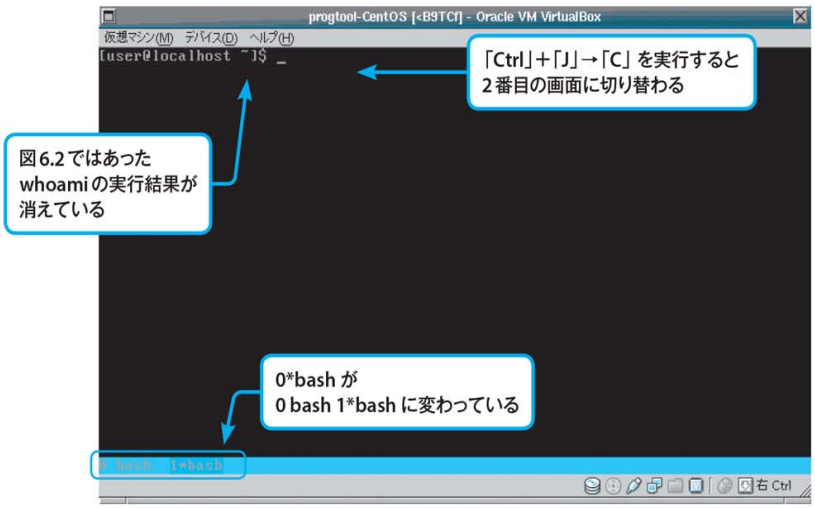


図6.3: スクリーンを追加する

あまり変化が無いようですが，**図6.3**では「whoami」の実行結果が消えてしまっています．

また最下部のバーに「bash」という欄がひとつ追加され，右の「bash」がマークされています．これは現在は2枚のスクリーンが生成され，2番目のスクリーンが選択されているということを示しています．

これは先ほどとは異なる新しいシェル環境なので，別に操作することができます． dateというコマンドで，時刻を表示してみましょう．

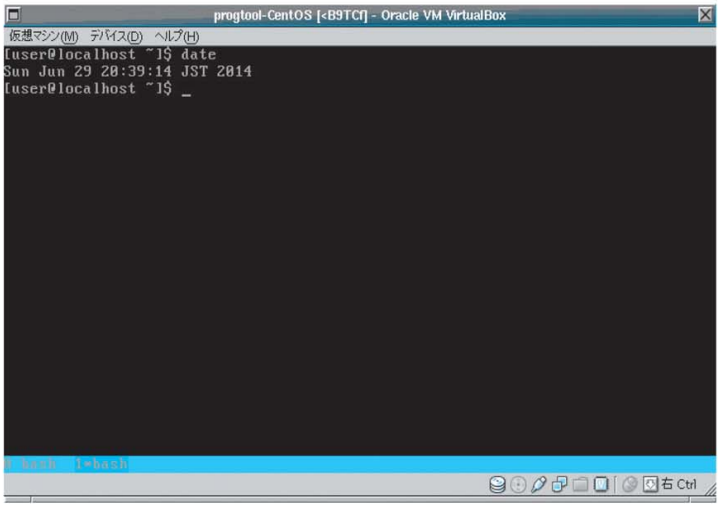


図6.4: dateコマンドを実行する

6.1.3 スクリーンを切り替える

この状態で，スクリーンを切り替えてみます．

スクリーンの切り替えはエスケープキーに続けて「P」か「N」です．「P」でひとつ前のスクリーンに戻り，「N」でひとつ先のスクリーンに進みます．とはいっても今は2枚のスクリーンしか開いていないので，どちらでも同じことにはなります．（先頭と末尾は繋がっており，循環します）

試しに「Ctrl」＋「J」→「N」のようにキーを押してみましょう．先ほどと同様に，「N」を押すときにはCtrlキーは押したままでも，離してしまってもかまいません．

すると，**図6.5**のようになります．

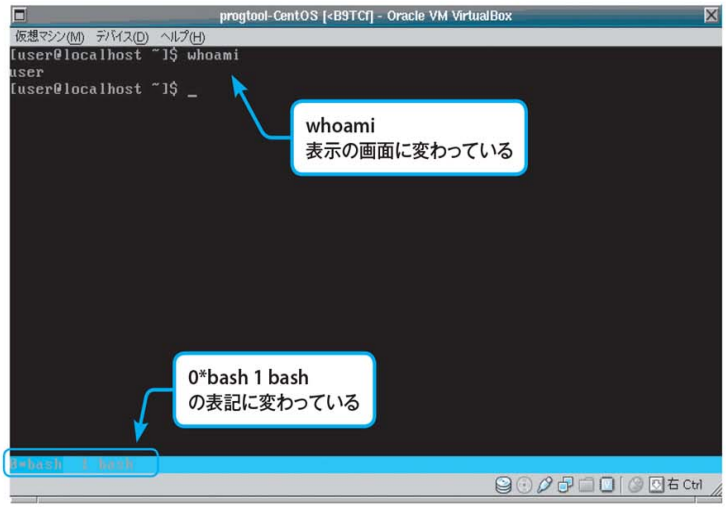


図6.5: スクリーンを切り替える

図6.5では，whoamiを実行した形跡が残っています．つまり，最初のシェル環境に戻っているようです．ここでもう一度「Ctrl」＋「J」→「N」を押すと，dateを実行したシェルに戻ります．

このようにしてスクリーンを次々に増やして，自由に切り替えながら利用することができます．

なおscreenコマンドのエスケープキーについてですが，先述したように本書のＶＭ環境では「Ctrl」＋「J」に設定してあります．

しかし実はscreenコマンドは，未設定のデフォルト状態ではエスケープキーは「Ctrl」＋「A」に割り当ててあります．

ただこれはシェルではカーソルの行頭への移動とぶつかってしまうため，変更しておく場合が多いようです．本書のＶＭ環境では「Ctrl」＋「J」に変更しているわけですが，このようにscreenコマンドのエスケープキーは環境によって異なることが考えられます．他の環境でscreenコマンドを利用する場合には，注意してください．

6.2 文字列をコピーしてみよう

screenコマンドでは，文字列をコピーすることができます．

コピーはスクリーン間でも行えます．これは，例えばシェルの実行結果をテキストエディタに張り付ける，逆にテキストファイル上のコマンド記述をコマンドラインに張り付ける，などといったときに便利です．

6.2.1 コピーモードに入る

コピーを行うにはまずコピーモードに入り，コピーする部分を選択する必要があります．

コピーモードには，エスケープキーに続けて「[」を押すことで入ることができます．つまり本書の環境では，「Ctrlキー」＋「J」→「[」のように押します．「[」のキーは，キーボードの右のほうにあります． Shiftキーなどといっしょに押す必要は無く，そのまま押してください．

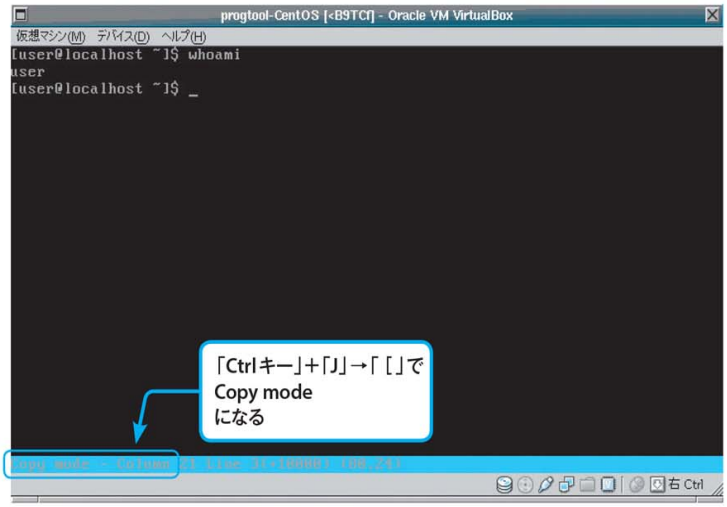


図6.6: コピーモードに入る

図6.6はコピーモードに入ったところです．

一見して，とくに変化は無いようにも見えます．しかしよく見ると，画面最下部のバーに「Copy mode」と表示されています．

コピーモードでは，矢印キーでカーソルを上下に移動することができるようになっています．試しに矢印キーの「↑」を押してみましょう．

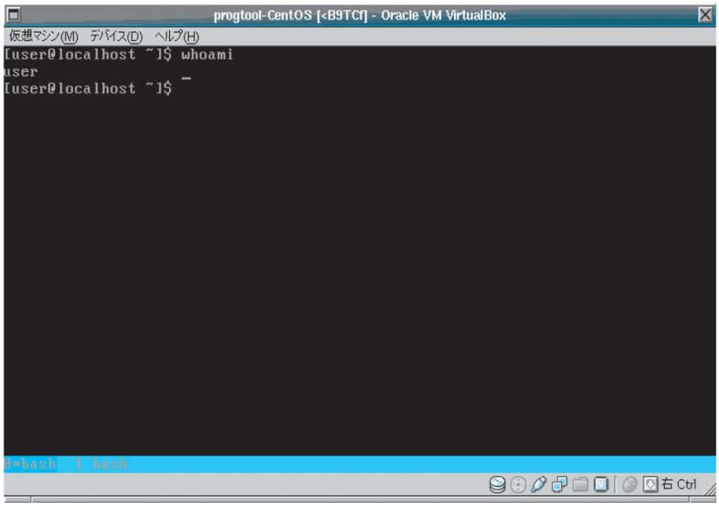


図6.7: 矢印キーの「↑」を押す

図6.7のようになり，カーソルがひとつ上に移動しています．シェルの操作では矢印キーの「↑」は履歴の表示でしたから，コピーモードに入ることで，それとは違った動作になっているわけです．

6.2.2 コピーする部分を選択する

この状態で矢印キーでカーソルを操作して，コピーしたい部分の先頭にカーソルを合わせてスペースキーを押します．

ここでは「whoami」の先頭の「w」の位置にカーソルを移動して，スペースキーを押してみましょう．

すると**図6.8**のようになり，「w」の一文字がハイライトされました．

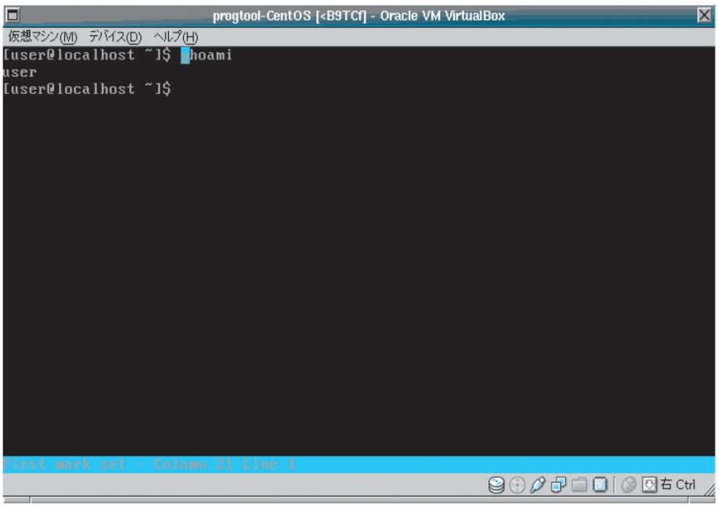


図6.8: コピーしたい部分の先頭でスペースキーを押す

さらに矢印キーの「→」を数回押して，「whoami」の全体を選択してみましょう．

すると**図6.9**のように，「whoami」の全体がハイライトされた状態になります．

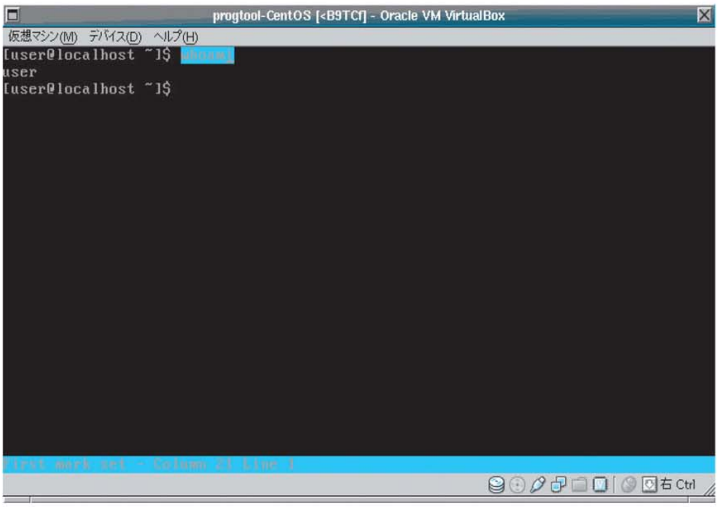


図6.9: コピーしたい部分の全体をハイライトする

ここでもう一度，スペースキーを押します．すると**図6.10**のようになり，「whoami」の全体が選択されたことになります．

これでコピー範囲の選択は完了し，コピーモードは終了しています．**図6.10**を見ると，カーソルはコピーモードに入る前の状態に戻っています．また最下部のバーには「Copied 6 characters」と出ていますが，これは選択された6文字がコピー用のバッファにコピーされた，という意味です．

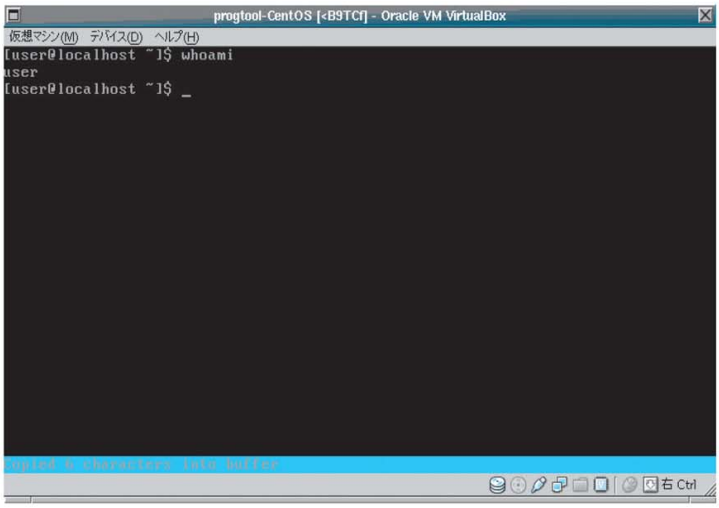


図6.10: スペースキーを押して，「whoami」の全体を選択する

6.2.3 コピー部分をペーストする

コピーモードで選択した部分はコピーモードから抜けた後に，「Ctrl キー」＋「J」→「]」のキーでペーストできます．「]」のキーは，「[」のキーの下にあります．これも Shift キーなどといっしょに押す必要な無く，そのまま押します．

ペーストすることで**図6.11**のようになります．「whoami」という文字列が，コマンドライン上にコピーされています．

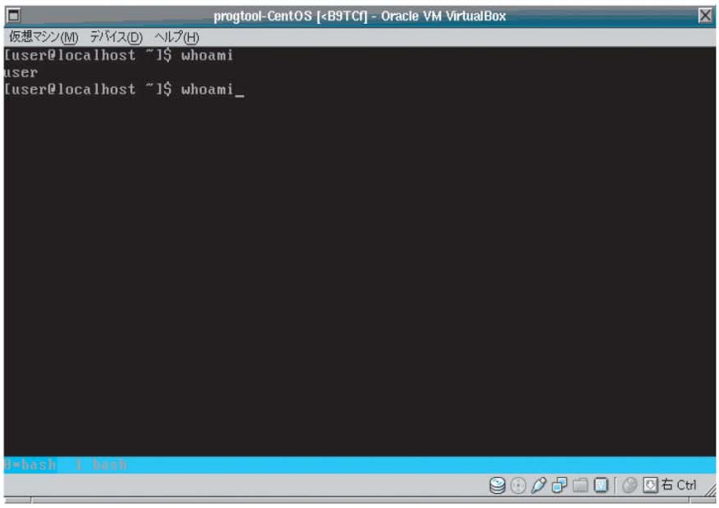


図6.11: コピーした部分をペーストする

ペーストは何度でもできますし，スクリーンを切り替えた後に行うこともできます．

たとえばここで「Ctrl」＋「J」→「N」を押してみます．画面は**図6.12**のようにして，２番目のスクリーンに切り替わります．

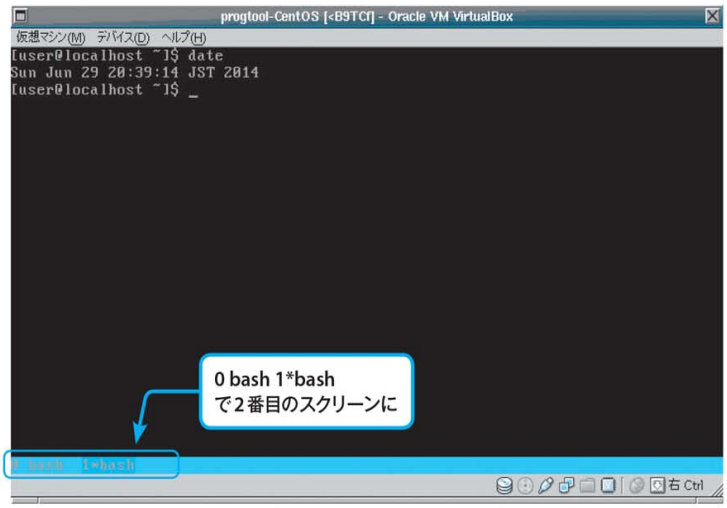


図6.12: ２番目のスクリーンに切り替える

「Ctrl」＋「J」→「]」で，ペーストしてみましょう．

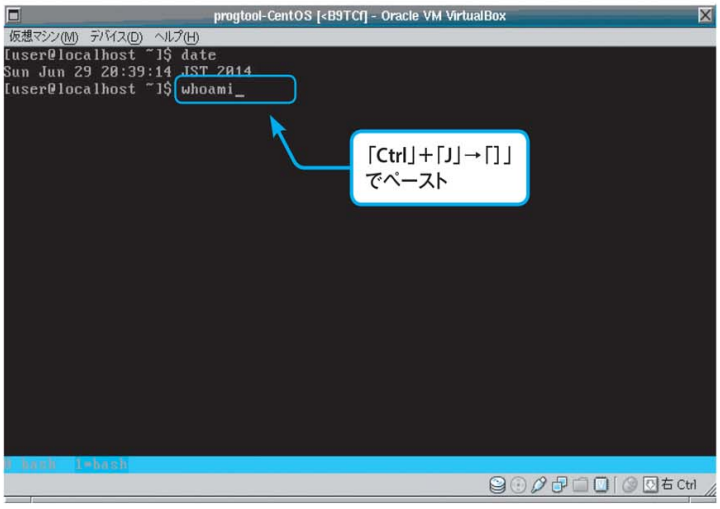


図6.13: スクリーンをまたいでペーストする

図6.13のように「whoami」がペーストできました．スクリーンをまたいでも，問題無くコピーアンドペーストができたようです．

6.3 まとめ

実は本書のＶＭ環境ではホームディレクトリに「.screenrc」という隠しファイルがあり，screenコマンドの操作は .screenrc を編集することでチューニングできます．（ファイル名の先頭にピリオド（「.」）が付きます）

本書のＶＭ環境ではエスケープキーを「Ctrl」＋「J」に割り当てていますが，これも変更することができます．インターネットなどで検索するといろいろ情報が出てきますので，.screenrc を自分好みに仕上げてみるといいでしょう． .screenrc は nano で編集することができます．

screenコマンドはブラインドタッチと非常に相性が良く，うまくチューニングして手に馴染ませると，キーボードから指を離す必要が無くなりたいへん便利です． GUIのようにいちいちマウスを動かす必要も無いため，慣れると手放せなくなるツールのひとつでもあります．

7

変数を使ってみよう

[C言語プログラミング編2]―変数

screenコマンドの使いかたを覚えたところで，C言語のプログラミング体験に戻りましょう．

コンピュータは「電子計算機」と呼ばれるように，その得意技は「計算」です．そして「数値」を扱うためには，数値の入れものである「変数」の使いかたを知る必要があります．

本章ではそんな「変数」を使って，数値計算を体験してみましょう．

7.1 変数を使うプログラムを書いてみよう

変数を使って，なにか簡単なプログラムを書いてみましょう．

プログラムをゼロから書いてもいいのですが，5章で作成した「hello.c」（**リスト5.2**）があるので，ここではそれをベースにすることで，ちょっと楽をしましょう．

まずは**リスト5.2**で作成した「hello.c」を「hello2.c」という名前にコピーし，nanoで開きます．

```
[user@localhost ~]$ cp hello.c hello2.c
[user@localhost ~]$ nano hello2.c
```

さらにhello2.cを，**リスト7.1**のように書き換えます．

なおhello.cをコピーしてhello2.cを作成しているのは，単に楽をしたいためだけの理由です．もしも「hello.c」が残っていなかったら，コピーせずにhello2.cを最初から書いて作成しても構いません．

リスト7.1: hello2.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(void)
005: {
006:         int a;
007:         a = 10;
008:         printf("a = %d\n", a);
009:         exit(0);
010: }
```

変更前の「hello.c」では「My name is ...」のようにして，printf() というもので名前を出力していました．**リスト7.1**の「hello2.c」ではこの部分を６～８行目のように変更しています．それ以外の「#include ...」や「int main...」，「exit...」などの行には，変更はありません．

hello2.c は「a」という「変数」を利用したプログラムの例です．コンパイルして実行してみましょう．

```
[user@localhost ~]$ gcc hello2.c -o hello2
[user@localhost ~]$ ./hello2
a = 10
[user@localhost ~]$
```

「a = 10」と表示されました．

7.1.1 変数に数値を代入しよう

リスト7.1のプログラムの意味を説明しましょう．

まず，次のような行があります．

```
        int a;
```

これは「a」という「変数」をこれから利用する，という意味です．C言語では変数を利用する場合には，このようにして「これから利用する」と明示する必要があります．

変数は，数値を格納するための「箱」です．例えば「int a;」の次には，以下のような行があります．

```
        a = 10;
```

「=」は「代入」という意味で，左の変数に対して右の値を格納することを示します．

つまりこれは，「変数a」に対して「10」という数値を代入することになります．

また変数には，好きな名前をつけることができます．

ここではとりあえず「a」のような簡単な名前にしていますが，「value」のような単語を使ったり，「VALUE」のように大文字にしたりすることもできます．「Value」のように大文字と小文字を組み合わせることもできます．大文字と小文字は区別されるので，「value」「VALUE」「Value」の３つはすべて異なる変数として扱われます．

さらに「sample_value」や「value2」のようにして，アンダーバーや数字を名前に用いることもできます．「this_is_sample_value」のような，長い名前を用いることもできます．この場合は，以下のような使いかたになるでしょう．

```
int this_is_sample_value;
this_is_sample_value = 10;
printf("value = %d\n", this_is_sample_value);
```

ただし数字は変数名の先頭に用いることはできません．つまり，「2value」のような名前の変数を使うことはできません．

7.1.2 数値の内容を出力しよう

リスト7.1の続く以下の行では，変数aの値を出力します．

```
printf("a = %d\n", a);
```

printf()はここまでで何度か利用してきましたが，指定された文字列を出力します．C言語では文字列は「"」（ダブルクォーテーション）でくくることで表現されます．つまり上では，2つの「"」でくくられている「a = %d\n」がひとつの「文字列」として扱われます．

つまりこれは，一見すると「a = %d」という文字列を出力するように思えます．「\n」の部分は，改行することを表しています．

しかし実はprintf()で「%d」という文字列を出力しようとすると，その部分は後続の変数の値に置き換えられます．

つまりこれは，文字列中の「%d」の位置を，変数の値に置き換えて表示することになります．そして「a = %d\n」という文字列の後には，カンマ記号（「,」）で区切られて変数aが指定されています．

変数aには「10」という値が代入されています．結果として，以下のように出力されるわけです．

```
a = 10
```

このようにして，printf()を用いて変数の値を出力させることができます．

7.2 計算をさせてみよう

ここまでは変数に対して単に数値を代入するだけでしたが，変数は計算に利用することができます．

次は変数による計算を試してみましょう．

7.2.1 計算をするように変更しよう

まず hello2.c を hello3.c というファイルにコピーし，nano で開きます．

```
[user@localhost ~]$ cp hello2.c hello3.c
[user@localhost ~]$ nano hello3.c
```

さらに hello3.c を，**リスト 7.2** のようなプログラムに書き換えてみます．これも，単に楽をするために hello2.c を hello3.c にコピーしているだけなので，コピーせずに hello3.c をゼロから作成しても，もちろん構いません．

リスト 7.2: hello3.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(void)
005: {
006:         int a;
007:         int b;
008:         int c;
009:
010:         a = 10;
011:         b = 20;
012:         c = a + b;
013:
014:         printf("%d + %d = %d\n", a, b, c);
015:         exit(0);
016: }
```

リスト 7.2 はちょっと長めのプログラムになっています．また変数も，「a」だけでなく「b」「c」などのいくつかを利用しているようです．

hello3.cをコンパイルして実行してみましょう.

```
[user@localhost ~]$ gcc hello3.c -o hello3
[user@localhost ~]$ ./hello3
10 + 20 = 30
[user@localhost ~]$
```

hello3を実行することで,「10＋20＝30」と表示されています.

7.2.2 どのような計算が行われているのか?

リスト7.2の動作を説明しましょう.

まずhello3.cには,次のような行があります.

```
        int a;
        int b;
        int c;
```

これは「a」「b」「c」という3つの「変数」をこれから利用する,という意味になります.

次に,以下の行があります.

```
        a = 10;
        b = 20;
```

これは,変数aに対して「10」という数値を代入し,さらに変数bに対して「20」という数値を代入することになります.

さらに,以下の行があります.

```
        c = a + b;
```

これは,「変数aと変数bを加算した値を変数cに代入する」という意味になります.

C言語のプログラムは上から順に実行されます．よってこれらは，以下のような順序で実行されることになります．

1. **変数aに「10」という値を代入する**
2. **変数bに「20」という値を代入する**
3. **変数cに変数a(つまり10)と変数b(つまり20)を加算した値(つまり10＋20で30)を代入する**

結果として変数cには「30」という値が代入されることになります．計算結果は次の行で出力されることになります．

```
        printf("%d + %d = %d\n", a, b, c);
```

出力文字列には「%d + %d = %d\n」が指定されています．こんどは出力する文字列中の３箇所に「%d」があります．そして文字列の後には「a, b, c」のようにして３つの変数が，カンマ記号「,」で区切られて指定されています．

このため出力の際には，この３箇所にそれぞれ「変数a」「変数b」「変数c」の値が順番に入ることになります．するとこれは，「10＋20＝30」のように置き換わることになります．

結果として，「10＋20＝30」と表示されることになるわけです．

7.3 まとめ

変数は数値の格納場所です．数値の一時的な保存や，様々な値の受渡しに利用することができます．

ここでは簡単な計算に変数を利用するだけでしたが，「変数d」も定義して使ってみるなど試してみてもいいでしょう．また加算でなく減算やかけ算など，様々な計算を試してみるのもいいでしょう．

また変数は，この後にC言語の「条件分岐」や「ループ」といったものを試していく中で，様々に利用されます．他のプログラミング言語でも，必ずといっていいほど使われる重要なものでもあります．

なお本章で利用したhello.c, hello2.c, hello3.cの３つのファイルは次章で「パッチ作成」の実験のために利用しますので，そのまま削除せずに置いておいてください．

8

パッチを作ってみよう
[ツール編3]—diff/patchコマンド

プログラムに対して修正を加えたとき，その修正内容を知りたくなることがあります．古いファイルと新しいファイルの間の差分を出力するコマンドを使えば，修正内容を「ファイルの差分」として得ることができます．

またこれを利用して，元のソースコードに対して差分を修正として加えることで，新しいソースコードを得ることができます．これが「パッチ」と呼ばれるものです．

本章ではそうした差分の見かた，そして「パッチ」の作り方と当てかたについて説明しましょう．

8.1 diffコマンドでファイルの差分を見てみよう

5章と7章では，「hello.c」「hello2.c」「hello3.c」という3つのファイルを作成しました．

これらを使って，ファイルの差分の取得を試してみましょう．まずこれら3つのファイルは，それぞれ**リスト8.1**，**リスト8.2**，**リスト8.3**のようになっていたはずです．

リスト8.1: hello.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(void)
005: {
006:         printf("My name is SAKAI Hiroaki.\n");
007:         exit(0);
008: }
```

リスト8.2: hello2.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(void)
005: {
006:         int a;
007:         a = 10;
```

```
008:         printf("a = %d\n", a);
009:         exit(0);
010: }
```

リスト8.3: hello3.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(void)
005: {
006:         int a;
007:         int b;
008:         int c;
009:
010:         a = 10;
011:         b = 20;
012:         c = a + b;
013:
014:         printf("%d + %d = %d\n", a, b, c);
015:         exit(0);
016: }
```

これらのファイルは部分的に似た箇所があります．例えばファイルの先頭４行はどのファイルでも「#include ～」や「int main ～」のようになっていて同じですし，終端が「exit(0)」で終了している点も一致しています．

このため7章では，hello2.c は hello.c に，そして hello3.c は hello2.c に部分的に変更を加えることで作成しました．

例えば hello2.c は，hello.c に対して以下のような修正を加えたものだということができます．

- **「printf()」の行は削除（リスト8.1の６行目）**
- **「int a;」という行を追加（リスト8.2の６行目）**
- **「a = 10;」という行を追加（リスト8.2の７行目）**
- **「printf("a = %d\n", a);」という行を追加（リスト8.2の８行目）**

このような「変更点」のことを，ファイルの「差分」と呼びます．

8.1.1 ファイルの差分の使い道は?

例えばあるファイルに対して，修正版のファイルを作成するような場合のことを考えてみましょう．

ファイルのサイズが小さい場合には，修正前のファイルはとくに使わずに，修正版のファイルをゼロから作成してしまったほうが早いかもしれません．

しかしサイズが巨大なファイルだったりすると，話は変わってきます．

修正前のファイルが数千行もあるような内容のものだったとして，その中で1行だけ変更を加えるような場合にはどうでしょうか？　そのようなときには修正前のファイルに対して，その変更したい行を修正するだけで，修正版のファイルを作成することができます．

修正版のファイルを他人に渡したい場合も，そうです．例えばその引渡し先のほうが修正前のファイルを持っているのならば，修正版のファイルをすべて送るのではなく，変更したい行の情報だけを送れば，修正版のファイルを生成することができるはずです．
これは巨大なファイルの全体を送付することよりも，ずっと楽でしょう．

しかし修正内容を正確に伝えることは，それはそれで面倒だとも思えます．例えば上記のhello.cとhello2.cの変更点を他人に説明するとしたら，どうすればいいでしょうか？

上に書いたように，「ファイルの差分」を言葉で表現して伝えるしかないものでしょうか？

8.1.2 diffコマンドで差分を出力してみよう

実はそのような「ファイルの差分」を出力してくれる「diff」というコマンドがあります．diffは「ディフ」と読みます．

以下のコマンドを実行してみましょう．

```
[user@localhost ~]$ diff -u hello.c hello2.c
```

すると，**リスト8.4**のような出力が得られるはずです．

リスト8.4: diffコマンドの出力（hello.c→hello2.c)

```
001: --- hello.c     2014-07-17 20:53:56.669090000 +0900
002: +++ hello2.c    2014-07-17 20:53:56.669090000 +0900
003: @@ -3,6 +3,8 @@
004:
005:  int main(void)
006:  {
007: -       printf("My name is SAKAI Hiroaki.\n");
008: +       int a;
009: +       a = 10;
010: +       printf("a = %d\n", a);
011:         exit(0);
012:  }
```

リスト8.4を見ると，行頭に「-」や「+」があったり，何もなかったりする行があるようです．

どうも，行頭に「-」がある行は「削除された行」，行頭に「+」がある行は「追加された行」を表しているようです．まとめると，以下のようになっています．

- **行頭に「-」のある行は，「その行は削除された」という意味になる**
- **行頭に「+」のある行は，「その行は追加された」という意味になる**
- **行頭に「-」も「+」も無い行は，変更無しでそのまま**

また出力は，変更がある行の前後数行のみが対象になっていて，それ以外の変更が無い行は省略されているようです．

例えば１万行もあるようなファイルで，１行しか変更されていないような場合があるとします．このようなとき，ファイルの全体を渡したりするのは無駄なことです．diffコマンドならば，このような変更は１行のみの変更として，**リスト8.4**のように数行で表現されるわけです．

hello2.c と hello3.c の差分も見てみましょう．

```
[user@localhost ~]$ diff -u hello2.c hello3.c
```

出力は**リスト8.5**のようになりました．

リスト8.5: diffコマンドの出力（hello2.c→hello3.c）

```
001: --- hello2.c    2014-07-17 20:53:56.669090000 +0900
002: +++ hello3.c    2014-07-17 20:53:56.669090000 +0900
003: @@ -4,7 +4,13 @@
004:  int main(void)
005:  {
006:          int a;
007: +        int b;
008: +        int c;
009: +
010:          a = 10;
011: -        printf("a = %d\n", a);
012: +        b = 20;
013: +        c = a + b;
014: +
015: +        printf("%d + %d = %d\n", a, b, c);
016:          exit(0);
017:  }
```

行頭が「+」の行は追加された行ですから，こちらは多くの行が追加されているようです．よく読むとprintf()による出力が修正され，変数b,cの定義や加算が追加されていることがわかります．

8.2 patchコマンドでパッチを当ててみよう

diffコマンドでファイルの差分を生成することができるわけですが，この差分は，ただ見て変更内容を確認するためだけにしか使えないというわけではありません．

例えば hello.c というファイルと，さらに hello.c から hello2.c への差分があれば，それらを合成して hello2.c を生成することができます．

8.2.1 「パッチ」とは何か?

「ファイルの差分」は修正後のファイルに対して，修正前のファイルの内容を差し引いたものと言えます．

ということは，以下のような式が成り立つということです．

「修正後」－「修正前」＝「差分」

diffコマンドはこのようにして差分を出力します．すると，以下のようなことも言えるはずです．

「修正前」＋「差分」＝「修正後」

つまり修正前のファイルに対して差分を加えることで，修正後のファイルを得ることができるわけです．

これを行うのが「patch」というコマンドです．「patch」は「パッチ」と読みます．

そしてこのような作業を「パッチを当てる」と言います．

8.2.2 パッチを当ててみよう

パッチ当てを試してみましょう．

まず，hello.c と hello2.c の差分を diff コマンドで取得します．

```
[user@localhost ~]$ diff -u hello.c hello2.c > hello.patch
```

このような差分ファイルを「パッチ」と呼びます．これは hello2.c ー hello.c →hello.patch のようにして，差分である「パッチ」を取得していると言えます．

次にpatchコマンドを使って，hello.c にパッチを当ててみましょう．パッチはシェルのリダイレクトである「<」を利用して，patchコマンドに与えます．リダイレクトについては3章を参照してください．

```
[user@localhost ~]$ patch < hello.patch
patching file hello.c
[user@localhost ~]$
```

これで hello.c にパッチが当てられ，hello.c の内容が更新されました．見てみましょう．

```
[user@localhost ~]$ cat hello.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        int a;
        a = 10;
        printf("a = %d\n", a);
        exit(0);
}
[user@localhost ~]$
```

hello.c の内容が，hello2.c と同じものに更新されています．パッチがうまく当たっているようです．

つまり hello.c ＋ hello.patch → hello2.c のように，ファイルの内容が更新されているわけです．

8.2.3 パッチの結果をdiffで確認しよう

これで hello.c は hello2.c と同じものに更新されたので，hello.c と hello2.c の内容は同一のはずです．

diffコマンドで，hello.cとhello2.cの内容を比較してみましょう．

```
[user@localhost ~]$ diff -u hello.c hello2.c
[user@localhost ~]$
```

差分が何も表示されません．ということは hello.c と hello2.c は同一の内容になっているということです． diffコマンドはこのように，ファイルの一致の確認に利用することもできます．

このような「パッチ」の仕組みは，数千行もあるようなファイルに変更を加える場合に便利です．

例えばファイルを新しいものに更新したい場合，そのファイル自体を渡すとしたら，数千行のファイルを渡さなければならないことになります．

パッチを使えば，変更点の部分だけの小さなパッチファイルを渡すだけで済むわけです．

8.2.4 パッチの結果を元に戻してみよう

パッチを逆に当てて，ファイルを元に戻すこともできます．

patch コマンドに -R というオプションを付加して実行してみましょう．

```
[user@localhost ~]$ patch -R < hello.patch
patching file hello.c
[user@localhost ~]$
```

これで hello.c の内容が元に戻ります．

確認してみましょう．

```
[user@localhost ~]$ cat hello.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        printf("My name is SAKAI Hiroaki.\n");
        exit(0);
}
[user@localhost ~]$
```

hello.c の内容が，元の状態に戻っているようです．

8.3 まとめ

今回はC言語のプログラムに対して差分を取得したりパッチを当てたりしてみましたが，これはプログラミング以外にも応用ができます．

例えば設定ファイルがテキスト表記の場合には，その変更内容をdiffで確認したりすることができます．ホームページの修正などに応用することもできるでしょう．

また単なる文章でも，テキスト形式で書いておけば，後でdiffコマンドで差分を得ることができます．これはドキュメントや仕様書などを書く場合に有用でしょう．WordやExcelファイルなどではできない，テキストファイルならではの便利な点とも言えます．

9

条件分岐をしてみよう
[C言語プログラミング編3]—if文

ここまでで作成したC言語のプログラムは，「書かれたものが上から順に実行されるだけ」というものです．

しかしこれだけでは複雑な処理はできません．決められたことを順次実行するだけだからです．

そこで次は「条件分岐」というものを試してみましょう．

9.1 if文での条件分岐を試してみよう

条件分岐とは，ある条件によってその後の処理が分かれるようなもののことです．

例えば「変数aの値が5より小さいときだけ，この処理を実行する」「変数bの値が10より大きければAの処理を，そうでなければBの処理を実行する」などといった具合です．

このように条件に応じて実行／非実行を切り替えることで，様々な複雑な処理を行うことができます．

9.1.1 if文の文法は?

C言語には条件分岐として「if文」という文法があります．

例えばif文を使って，以下のように書くことができます．

```
if (条件) {
  ...条件が満たされた場合には，ここを実行する
} else {
  ...そうでなかった場合には，ここを実行する
}
```

このように条件が満たされたときの処理や，そうでなかったときの処理を，「{…}」でくくって表記します．「{…}」の中には，複数の処理を書くことができます．

「そうでなかった場合」は省略することもできます．その場合には，次ページのようになるでしょう．

```
if（条件）{
  ...条件が満たされた場合には，ここを実行する
}
```

「{…}」のように中括弧でくくられている部分を「ブロック」と呼びます．

つまりif文は，条件が満たされた場合には直後のブロックの中の処理を実行し，そうで無かった場合には，（存在するならば）「else」以降のブロックの中の処理を実行します．

例えば以下のように書かれているとします．

```
処理A
if（条件）{
  処理B
} else {
  処理C
}
処理D
```

条件が満たされる場合には，A→B→Dのように実行されます．満たされない場合には，A→C→Dのように実行されます．

またブロックの中には，さらにif文を書くこともできます．例えば以下のような具合です．

```
処理A
if（条件1）{
  処理B
  if（条件2）{
    処理C
  } else {
    処理D
  }
  処理E
} else {
  処理F
}
処理G
```

上の例では，条件１と条件２に応じて，以下のように実行されます．

- **条件１と条件２が満たされる場合　A→B→C→E→G**
- **条件１が満たされ，条件２が満たされない場合　A→B→D→E→G**
- **条件１が満たされない場合　A→F→G（条件２は影響しない）**

9.1.2 if文を使ってみよう

実際にif文を使って，**リスト9.1**のようなプログラムを書いてみましょう．ファイル名はif.cとします．

リスト9.1: if.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(int argc, char *argv[])
005: {
006:         printf("Start! argc = %d\n", argc);
007:         if (argc < 3) {
008:                 printf("Apply\n");
009:         } else {
010:                 printf("Not apply\n");
011:         }
012:         printf("End!\n");
013:         exit(0);
014: }
```

リスト9.1では4行目のmain()の括弧の中に「int argc, char *argv[]」という謎の表現が出てきています．

これは新しい書き方ですが，とりあえずは気にせず，以下のことだけ覚えてください．

「プログラムを実行したときの引数の個数＋1が，argcという変数に格納される」

ここで「引数」と言っているのは，「コマンドライン引数」のことです．

if文による条件分岐を試すためには，条件を変更できるようなパラメータが必要です．このためには実行時に，プログラム側で何らかの入力を受け取ることが必要です．ここでは「コマンドライン引数の数」を受け取って変更しながら試すことで，条件分岐の実験をしてみます．

9.1.3 さまざまな入力で実行してみよう

if.c をコンパイルして，実行してみましょう．

```
[user@localhost ~]$ gcc if.c -o if
```

まず，以下のようにして単純に実行してみます．

```
[user@localhost ~]$ ./if
Start! argc = 1
Apply
End!
[user@localhost ~]$
```

リスト9.1では6行目で，変数argcの値を表示しています．実行結果では「argc = 1」と出力されているため，変数argcは「1」という値になっていることがわかります．

変数argcにはコマンドライン引数の個数＋1の値が格納されますが，コマンドライン引数無しで実行しているため，「1」となっているようです．その後には「Apply」と出力され，さらに「End!」と出力されてプログラムは終了しています．

次に，以下のように適当なコマンドライン引数を持たせて実行してみましょう．

```
[user@localhost ~]$ ./if aaa
Start! argc = 2
Apply
End!
[user@localhost ~]$
```

「argc = 2」と出力されています．つまり変数argcの値は「2」になっています．「Apply」「End!」の出力は変わっていません．

今度はコマンドライン引数に「aaa」がひとつだけ指定されています．このため変数argcは，コマンドライン引数の個数＋1として，「2」という値になっているわけです．

9.1.4 引数を2つ指定してみよう

次に，以下のように２つのコマンドライン引数を指定して実行してみます．

```
[user@localhost ~]$ ./if aaa bbb
Start! argc = 3
Not apply
End!
[user@localhost ~]$
```

今度は実行結果が変わっています．ちょっと説明しましょう．

まず，出力は「argc = 3」のようになっています．つまり変数argcの値は「３」です．実行時にコマンドライン引数に「aaa」に加えて「bbb」も指定されているため，コマンドライン引数の数は「２」です．このため変数argcには，２を＋１した「３」という値が格納されているようです．

次に，今までは「Apply」と出力されていたものが，今度は「Not apply」と出力されています．

これは**リスト9.1**の７行目にあるif文の効果です．

１度目と２度目の実行のときは，コマンドライン引数の数はゼロと１個でした．このため変数argcの値は「１」と「２」でした．

さらに**リスト9.1**の７行目のif文の条件は，「argc < 3」のようになっています．つまり「変数argcが，３より小さければ」という条件です．

変数argcの値が１と２のときには，この条件は満たされます．よってif文の直後のブロックである８行目が実行され，「Apply」と表示されます．

３度目の実行のときは，コマンドライン引数の数は２個で，変数argcの値は「３」でした．よってif文の条件は満たされないことになります．このためelse以降のブロックである10行目が実行され，「Not apply」と表示されるわけです．

では，コマンドライン引数の数が３個の場合はどうなるのでしょうか．

```
[user@localhost ~]$ ./if aaa bbb ccc
Start! argc = 4
Not apply
End!
[user@localhost ~]$
```

このときの変数argcの値は「４」です．このためif文の条件は満たされません．

よって3度目の実行のときと同様に，「Not apply」と表示されています．

9.1.5 条件を変えてみよう

ここで，ちょっと条件を変えてみましょう．

リスト9.1はif文の条件が「argc < 3」のようになっています．これを「argc == 3」のようにしてみましょう．

if.c を if2.c にコピーして，if文の条件を「argc == 3」のように変更します． if2.cは**リスト9.2**のようになります．

リスト9.2: if2.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(int argc, char *argv[])
005: {
006:         printf("Start! argc = %d\n", argc);
007:         if (argc == 3) {
008:                 printf("Apply\n");
009:         } else {
010:                 printf("Not apply\n");
011:         }
012:         printf("End!\n");
013:         exit(0);
014: }
```

前の章でdiffによる差分の確認を体験したので，せっかくなのでdiffを使って差分を表記してみましょう． if.c と if2.c の差分は，以下のようになります．

```
[user@localhost ~]$ diff -u if.c if2.c
--- if.c        2014-07-17 23:31:16.133129726 +0900
+++ if2.c       2014-07-17 23:31:16.134129734 +0900
@@ -4,7 +4,7 @@
 int main(int argc, char *argv[])
 {
        printf("Start! argc = %d\n", argc);
-       if (argc < 3) {
+       if (argc == 3) {
                printf("Apply\n");
        } else {
```

```
                printf("Not apply\n");
[user@localhost ~]$
```

if2.c を実行してみます．今度は引数の数をいろいろ変えながら，まとめて調べてみましょう．

```
[user@localhost ~]$ gcc if2.c -o if2
[user@localhost ~]$ ./if2 aaa
Start! argc = 2
Not apply
End!
[user@localhost ~]$ ./if2 aaa bbb
Start! argc = 3
Apply
End!
[user@localhost ~]$ ./if2 aaa bbb ccc
Start! argc = 4
Not apply
End!
[user@localhost ~]$
```

今度はコマンドライン引数を「aaa bbb」としたときだけ「Apply」となっています．他はすべて「Not apply」です．

リスト9.2の7行目のif文の条件は「argc == 3」のようになっています．これは「変数argcの値が，3と等しければ」という意味になります．よって引数を「aaa bbb」と2つ指定したときのみ，if文の条件が満たされるわけです．

C言語では「=」は変数への代入を表しますが，「==」は「一致していること」を表します．

また「<」は「少ないこと」を表しますが，「>」はその逆で，大きいことを表します．さらに「<=」は「以下」，「>=」は「以上」を表します．そして「!=」は「等しくないこと」を表します．

このように，様々な比較を行うことができるようになっています．

9.2 まとめ

if文を利用することで，プログラムを様々な条件で細かく分岐させて複雑な処理を行うことができます．

条件分岐が無ければ，プログラムは決められた計算をただ順番に実行することしかできません．これではただの「自動演算機」です．

しかし条件分岐という機能を持つことで，様々な条件に応じて多彩な処理を行うことができるようになっています．これがコンピュータを単なる演算機にとどまらせずに，「情報処理」を行う機械にしている要素のひとつだと言えるでしょう．

●本書で紹介するCentOSのCUIコマンド―2（3章）

コマンド	ツールの役割	ページ
less	ファイルビューワのコマンド	79
<	ファイルの先頭に一気にジャンプ	80
>	ファイルの末尾にジャンプ	80
/	検索	83
Q	lessコマンドの終了	84
cat > （ファイル名）（入力文字）	［リダイレクト］　キーボード入力の文字を（ファイル名）に書き込む	86
「Ctrl」+「D」	リダイレクトモードから抜ける	86
|	［パイプ］　コマンドの出力を別のコマンドの入力に繋げる	88
ls （ディレクトリ） | head -n 10	lsの結果で先頭10行のみ表示	88
ls （ディレクトリ） | tail -n 10	lsの結果で末尾10行のみ表示	88
ls （ディレクトリ） | grep ^a	lsの結果で「a」で始まる行のみを出力	91
ls （ディレクトリ） | grep ^a | less	lsの結果で「a」で始まる行のみをlessで閲覧する	91
cat （ファイル名） | sort	（ファイル名）の中身を降順に出力 ※1行を1件のレコードと見なす	92
cat （ファイル名） | sort -r	（ファイル名）の中身を昇順に出力	92
cat （ファイル名） | sort -n	（ファイル名）の中身を数字順に出力	93
cat （ファイル名） | sort | uniq	重複する行を1行にまとめる（uniqコマンドは「隣り合った重複行」をまとめる）	95
cat （ファイル名） | sort | uniq | wc -l	重複する行をまとめて行数をカウント	95
man （コマンド名）	（コマンド名）のマニュアルを表示する	96
Q	manコマンドの終了	97

10

ソースコードを管理しよう

[ツール編4]—gitコマンド

ここまででC言語のプログラムをいくつか書いてきました．そしてそれらは，hello.c, hello2.c, hello3.c のような適当なファイル名で扱ってきました．

しかしこのように，作成したプログラムを単にファイルとして置いておくだけでは，どれがどのファイルなのか，最新版はどれなのか，すぐにわからなくなってしまいます．

そのようなことを防ぐために，実際の開発の際には，プログラムのファイルは何らかの管理ツールで管理するのが常です．

またそうした管理ツールはソースコードの管理のみならず，ドキュメントや設定ファイルなどの管理にも応用できます．プログラマに限らず，知っておいて何かと得だと思います．

ここでは「git（ギット）」という管理ツールを体験してみます．

10.1 gitリポジトリを作成してみよう

ソースコードの管理には，古くは「CVS」というツールが使われてきました．現在ではその後継である「Subversion」が広く利用されています．

「git」は比較的新しい管理ツールですが，Linuxのソースコード管理のために生まれたということもあり，またSNSのような情報交換の仕組みを加えた「github」の人気もあって，普及しているツールです．正式には「ギット」と発音しますが，「ジット」と言っても通じるでしょう．

リポジトリが簡単に作れるというメリットもあるので，本書ではgitを使ってみましょう．

10.1.1 初期設定をしよう

gitを利用する前に，初期設定をします．これは最初に一度だけ行えば大丈夫です．

まずは以下のようにして，ユーザ名とメールアドレスを登録します．以下は筆者の例ですが，読者のかたは名前などを適宜変更して登録してください．なおメールアドレスは実際にメールが出されるわけではないため，適当なものでも構いません．

```
[user@localhost ~]$ git config --global user.name "SAKAI Hiroaki"
[user@localhost ~]$ git config --global user.email hiroaki@mail.address
```

さらに標準で使用するテキストエディタを指定します．

エディタには，とりあえず nano を指定しておけばいいでしょう．これは未指定だとviエディタが使われることになります．

```
[user@localhost ~]$ git config --global core.editor nano
```

設定内容を確認しておきましょう．以下のようにgitコマンドを利用することで，設定内容を出力させることができます．

```
[user@localhost ~]$ git config --list
user.name=SAKAI Hiroaki
user.email=hiroaki@mail.address
core.editor=nano
[user@localhost ~]$
```

10.1.2 リポジトリを作成しよう

設定が終わったところで，gitによるファイル管理を体験してみましょう．

まずはファイル管理のための「リポジトリ」を作成します．リポジトリとは，ファイルを管理するためのデータベースのことです．

データベースというとなんだか難しそうですが，作り方は簡単です．適当なディレクトリを作成し，その中で「git init」というコマンドを実行するだけです．

ここでは「git」というディレクトリを作成して，そこをリポジトリにしてみましょう．

```
[user@localhost ~]$ mkdir git
[user@localhost ~]$ cd git
[user@localhost git]$ git init
Initialized empty Git repository in /home/user/git/.git/
[user@localhost git]$
```

これでgitというディレクトリがリポジトリとして利用できるようになりました．

何か変化があったのでしょうか．lsで確認してみましょう．

```
[user@localhost git]$ ls
[user@localhost git]$
```

これではとくに何も変化が無いように見えますが，以下のように「ls」に「-a」を付加して実行すると，「.git」というディレクトリが生成されていることがわかります．

```
[user@localhost git]$ ls -a
.  ..  .git
[user@localhost git]$
```

この .git というディレクトリはファイルの管理にgitコマンドが使うものですので，内容を知る必要はありません．とりあえず，うっかり消さないように注意しておけばいいだけです．

なお先頭に「.」(ピリオド) が付加されているファイルやディレクトリは，「ls」では表示されません．このため設定ファイルなど，普段は表示しなくてもいいようなファイルのファイル名として用いられます．なお「ls」で表示されないだけで，それ以外のファイル操作は普通に可能です．

10.1.3 ファイルをリポジトリに登録しよう

次に5章で作成したhello.c（**リスト5.1**）を，リポジトリに登録してみましょう．

まずはリポジトリ上に，ホームディレクトリにあるhello.cをコピーします．

```
[user@localhost git]$ cp ../hello.c .
```

もしもhello.cを削除してしまった場合には，nanoで作成しなおしましょう．hello.cは以下のような内容になっています．

```
[user@localhost git]$ cat hello.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        printf("My name is SAKAI Hiroaki.\n");
        exit(0);
}
[user@localhost git]$
```

hello.cをコピーしたら，これをgitに登録します．登録はまず git add を行い，さらに git commit することで行います．

```
[user@localhost git]$ git add hello.c
[user@localhost git]$ git commit hello.c
```

これで登録は完了です．このように登録することを，「ファイルをコミットする」と言います．

10.1.4 コメントを入力しよう

git commit を実行すると，**図 10.1** のようにテキストエディタが開きます．初期設定でエディタに nano を指定しているので，nano が起動しているはずです．

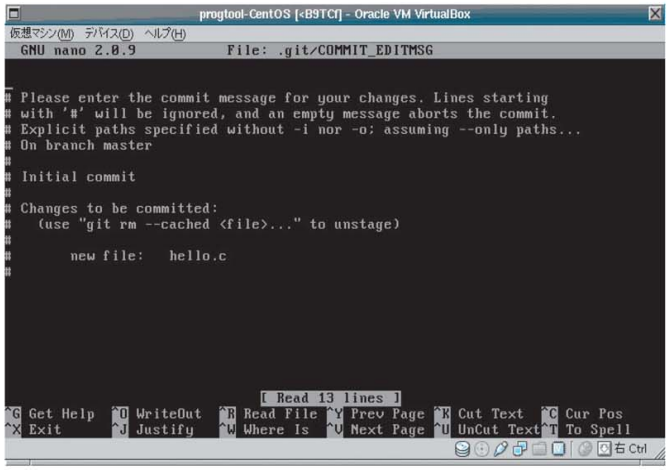


図 10.1: git commit を実行する

これは git がコメントの入力を求めてきているためです．コメントは，ファイルのコミット時のログとして保存されます．

ひとまず「Sample of Hello World.」とでも入力しておきましょう．

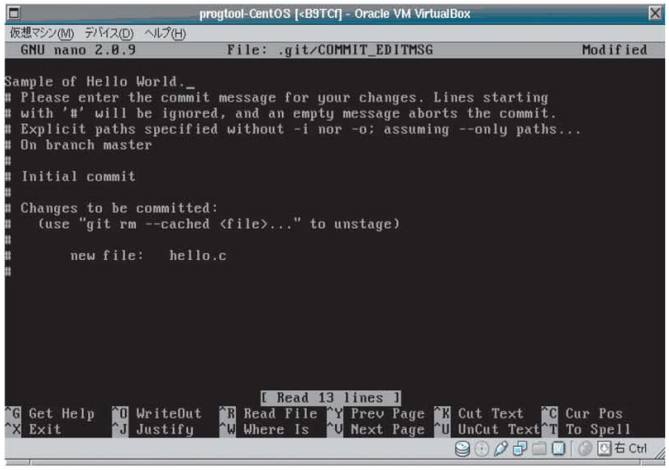


図 10.2: コメントの入力画面

入力したら，Ctrl+X を押してコメントを保存し終了します．保存していいかどうかを確認してきますので，「Y」を選択します．

すると**図10.3**のように「File Name to Write:」として保存のためのファイル名が出てきますが，表示されたファイル名のままEnterキーを押します．

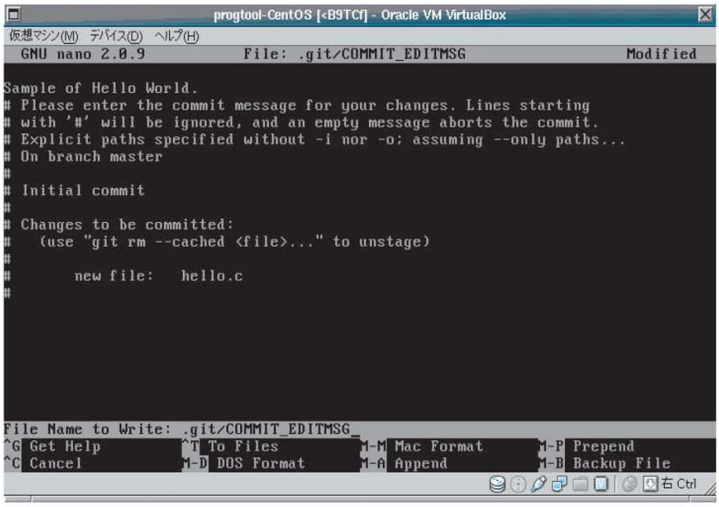


図10.3: コメントを保存する

コメントを保存するとコミットの処理は完了し，**図10.4**のようになります．

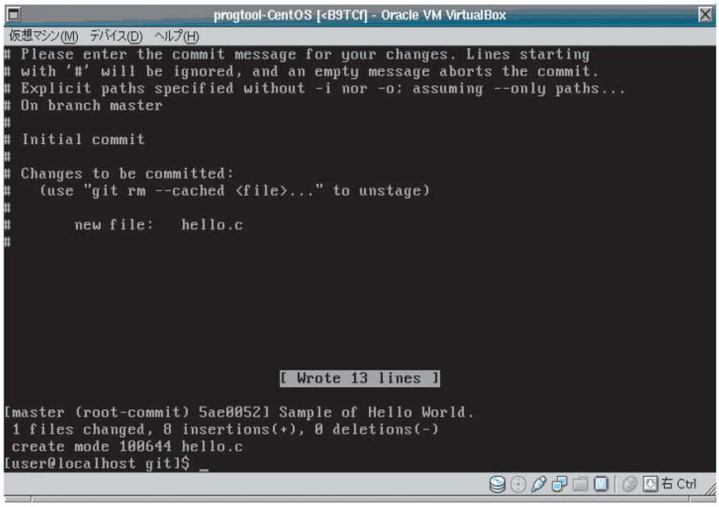


図10.4: git commit を完了した

これでコミットは完了です．

10.1.5 ログを確認しよう

これで hello.c はリポジトリに登録できました.

ファイルの履歴は git log で見ることができます.

```
[user@localhost git]$ git log hello.c
commit 5ae00526a8c325dd746e8b5c52b31e6a72733655
Author: SAKAI Hiroaki
Date:   Sat Jul 5 21:02:30 2014 +0900

    Sample of Hello World.
[user@localhost git]$
```

コミットしたときの日付がわかります．またコミット時に入力したコメント「Sample of Hello World.」が，ログとして残っています.

このようにgitが履歴管理をしてくれるため，コメントを適切に入力しておけば，どのような理由で登録されたファイルなのかがわかります.

10.2 ファイルを元に戻してみよう

hello.cは登録済みでgitによって管理されているため，たとえ削除してしまっても，いつでも取り戻すことができます．

例えばhello.cを間違えて削除してしまったとしましょう．

```
[user@localhost git]$ rm hello.c
[user@localhost git]$ ls
[user@localhost git]$
```

そしてhello.cを取り戻してみましょう．

10.2.1 削除したファイルを取り戻そう

ファイルを元に戻すのは「git checkout」です．やってみましょう．

```
[user@localhost git]$ git checkout hello.c
[user@localhost git]$ ls
hello.c
[user@localhost git]$
```

削除されたhello.cが，元に戻っています．

内容も確認してみましょう．

```
[user@localhost git]$ cat hello.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        printf("My name is SAKAI Hiroaki.\n");
        exit(0);
}
[user@localhost git]$
```

たしかにhello.cが元に戻っているようです．

10.2.2 ファイルを変更してみよう

次にhello.cに改造を入れてみます．エディタを開いてhello.cに適当に何かを追加してみましょう．

```
[user@localhost git]$ nano hello.c
```

ここでは「Hello World!」と出力している行を以下のように修正し，「HELLO WORLD!」のようにすべて大文字にしてみました．

```
        printf("HELLO WORLD!\n");
```

ファイルを修正してnanoを終了すると，hello.cの内容は，以下のように更新されました．

```
[user@localhost git]$ cat hello.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        printf("HELLO WORLD!\n");
        exit(0);
}
[user@localhost git]$
```

hello.cの変更点を見てみましょう．ファイルに変更を加えた場合，その変更点は「git diff」で確認できます．

```
[user@localhost git]$ git diff hello.c
diff --git a/hello.c b/hello.c
index 9d074c8..37dc91f 100644
--- a/hello.c
+++ b/hello.c
@@ -3,6 +3,6 @@

 int main(void)
 {
-       printf("Hello World!\n");
+       printf("HELLO WORLD!\n");
        exit(0);
 }
[user@localhost git]$
```

これはリポジトリに登録済みのhello.cに対して，現在のhello.cの差分を出力しています．出力のフォーマットは8章で説明したdiffコマンドと同様のものです．

行頭が「-」の行は削除された行で，行頭が「+」の行は追加された行です．つまり「Hello World!」と出力している行が削除され，さらに「HELLO WORLD!」を出力している行が追加されているわけです．出力文字列が，すべて大文字に変更されていることがわかります．

10.2.3 変更したファイルを元に戻そう

そして，やっぱりその改造はボツとして，元に戻したくなったとします．

ファイルを元に戻すのも，「git checkout」で可能です．

```
[user@localhost git]$ git checkout hello.c
[user@localhost git]$ git diff hello.c
[user@localhost git]$
```

git diff で何も表示されなくなっています．つまり，hello.c が元に戻っているということです．

内容も確認してみましょう．

```
[user@localhost git]$ cat hello.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        printf("Hello World!\n");
        exit(0);
}
[user@localhost git]$
```

確かに元に戻っているようです．

10.3 ファイルを更新してみよう

これで hello.c は登録されたわけですが，ファイルは一度登録すればそれで終わりではありません．プログラムのソースコードなどは，どんどん改良しては更新していくのが常です．

ファイルの更新も，git で管理することができます．次は hello.c に変更を加えて，更新してみましょう．

10.3.1 ソースコードに処理を追加しよう

まずすでに登録済みの hello.c をエディタで開き，適当に処理を追加します．

```
[user@localhost git]$ nano hello.c
```

ここでは「Hello World!」の表示の後に，以下のような名前の表示を追加してみました．名前はひとまず筆者のものにしておきましたが，読者のかたがたは自分の名前で試してみてください．

```
        printf("My name is SAKAI Hiroaki.\n");
```

修正後の hello.c の内容は以下のようになります．

```
[user@localhost git]$ cat hello.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        printf("Hello World!\n");
        printf("My name is SAKAI Hiroaki.\n");
        exit(0);
}
[user@localhost git]$
```

修正内容を確認してみましょう．git diff で修正の差分を見てみます．

```
[user@localhost git]$ git diff hello.c
diff --git a/hello.c b/hello.c
index 9d074c8..3888a63 100644
```

```
--- a/hello.c
+++ b/hello.c
@@ -4,5 +4,6 @@
 int main(void)
 {
        printf("Hello World!\n");
+       printf("My name is SAKAI Hiroaki.\n");
        exit(0);
 }
[user@localhost git]$
```

「My name is ...」を表示する行が追加されています．

10.3.2 ファイルを再登録しよう

修正内容を再登録してみましょう．手順は hello.c を初回で登録したときと同じで，「git add」「git commit」を続けて実行します．

```
[user@localhost git]$ git add hello.c
[user@localhost git]$ git commit hello.c
```

git commit を実行すると，初回と同じように**図 10.5** のようにして nano が起動し，コメント入力を求めてきます．

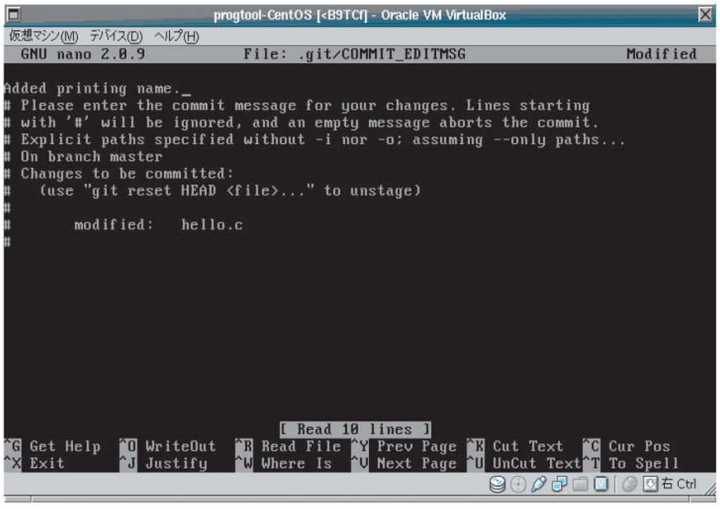


図10.5: ２度目の git commit を実行する

コメントは，今度は「Added printing name.」のように入力してみました．

入力したら，Ctrl＋X で nano を終了します．

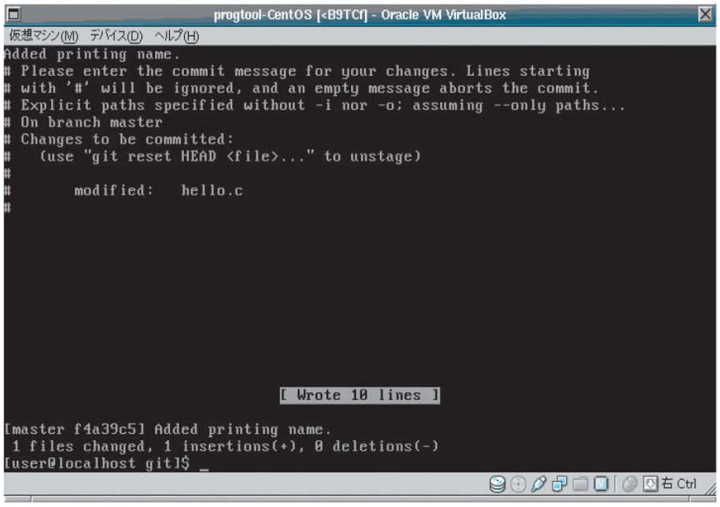


図10.6: ２度目の git commit を完了した

10.3.3 更新後のログを見てみよう

これで hello.c が，改良版に更新されました．

git log で，ファイルのログを見てみましょう．

```
[user@localhost git]$ git log hello.c
commit f4a39c5bb195d5eee37bec14c8032eb60277da7f
Author: SAKAI Hiroaki
Date:   Sat Jul 5 21:14:38 2014 +0900

    Added printing name.

commit 5ae00526a8c325dd746e8b5c52b31e6a72733655
Author: SAKAI Hiroaki
Date:   Sat Jul 5 21:02:30 2014 +0900

    Sample of Hello World.
[user@localhost git]$
```

２つの版のログが残っているようです．「Sample of Hello World.」というコメントのものと，「Added printing name.」というコメントのものです．

ログの登録時刻を見ると，「Added printing name.」のほうが後になっています．つまりこちらのほうが，最新版だとわかります．

hello.c を削除して，git checkout で取り出してみましょう．

```
[user@localhost git]$ rm hello.c
[user@localhost git]$ ls
[user@localhost git]$ git checkout hello.c
[user@localhost git]$ ls
hello.c
[user@localhost git]$
```

これで取り出した hello.c は，最新版のものになります．

内容も確認しておきましょう．

```
[user@localhost git]$ cat hello.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        printf("Hello World!\n");
        printf("My name is SAKAI Hiroaki.\n");
        exit(0);
}
[user@localhost git]$
```

「My name is ...」の行が追加されているので，きちんと最新版が取り出せているようです．

10.3.4 ファイルを移動しよう

さて，ここではgitリポジトリのトップディレクトリにファイルを格納しましたが，今後ファイルを追加していく際のことを考えると，ディレクトリを作成して分類しておいたほうがいいように思えます．

ということで「sample」というディレクトリを作成し，hello.c をそちらに移動しておきましょう．

```
[user@localhost git]$ mkdir sample
[user@localhost git]$ ls
hello.c  sample
[user@localhost git]$
```

ファイルの移動は「git mv」で行います．単にmvコマンドで移動するだけでは駄目なので，注意してください．

```
[user@localhost git]$ git mv hello.c sample
```

git mv を実行したら，hello.c が移動できているかどうかを確認しましょう．

```
[user@localhost git]$ ls
sample
[user@localhost git]$ ls sample
hello.c
[user@localhost git]$
```

ディレクトリ「sample」の中に移動できたようです．

しかし，これだけではまだ不十分です． git mv による移動後は，git commit で登録する必要があります．

```
[user@localhost git]$ git commit .
```

git commit の引数に「.」を指定するとカレントディレクトリを指定したことになり，カレントディレクトリ以下のファイルが登録対象になります．コメントは「Move file.」とでもしておけばいいでしょう．

10.4 まとめ

gitには他にも多彩な機能がありますが，とりあえずはファイルの登録と復旧，ログと差分の表示くらいを覚えておきましょう．あとは実際に使ってみて，必要に応じていろいろ覚えていけばいいでしょう．

gitは各種ドキュメントや設定ファイルの管理にも利用できます．つまり，プログラマに限らず知っておいて損は無いツールです．ぜひ活用したいところです．

またgitは共有リポジトリとして利用することで，複数人で作業をする際に威力を発揮します．

例えば複数人で同じファイルを扱う場合，ネットワークドライブを立てて共有するファイルはそこに置くような運用がされることが多々あります．しかしこれでは，誰かが修正中に他のひとが修正できなかったり，だれがいつ修正を入れたのかが不明確になったりといった問題がつきまといます．

gitを使うことでファイルの履歴が適切に管理され，このような問題を回避することができます．

11

ループを使ってみよう

[C言語プログラミング編4]－while/for文

プログラミングに重要な要素のひとつとして，9章では「条件分岐」というものを説明してみました．またC言語では「if文」というもので，条件分岐できることを試しました．

そしてプログラミングにはもうひとつ，重要な要素があります．それは「繰り返し」というものです．

11.1 while文でのループを試そう

プログラム中で，特定の処理を繰り返し実行したいということがあります．

例えば検索処理です．あるデータベースの中から指定のキーワードを検索するような場合，データベースの各エントリとの比較を繰り返し行うことで検索が行えます．

そのようなときに用いられるのが「ループ」です．ループは特定の処理を，特定の条件が満たされるまで繰り返します．上の「検索」の例では，キーワードが一致するまで比較を繰り返す，という処理を行えばいいわけです．

C言語ではループを行うために，「while文」と「for文」という文法があります．

11.1.1 while文を使ってみよう

まず，「while文」というものを使ってみましょう．

リスト11.1はwhile文を使ったサンプルです．これをloop.cという名前のファイルで作成してみてください．

リスト11.1: loop.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(int argc, char *argv[])
005: {
006:         while (1) {
007:                 printf("In loop.\n");
```

```
008:         }
009:         exit(0);
010: }
```

loop.c を作成したら，コンパイルして実行ファイルを作成します．

```
[user@localhost ~]$ gcc loop.c -o loop
```

ではこれを実行してみましょう…とその前に，ひとつ注意点がありますので説明しておきます．

このプログラムを実行すると，「In loop」というメッセージが延々と出力されます．このような現象は初めて遭遇すると，ちょっとあせってしまうかもしれません．

でも大丈夫です．Ctrl＋Cで止めることができますので，慌てないでください．

では，実行してみましょう．

```
[user@localhost ~]$ ./loop
In loop.
In loop.
In loop.
In loop.
...
```

説明したように「In loop」というメッセージが延々と出力されて，シェルのプロンプトはあっと言う間に画面外に流れ出てしまうでしょう．

これはそのままでは止まりませんので，「Ctrl キー」＋「C」を押すことでプログラムの実行を中断します．

```
In loop.
In loop.
In loop.
In loop.^C
[user@localhost ~]$
```

無事に中断できたでしょうか？中断できない場合にも，繰り返し押したりしてみてください．一度押せばしばらく待っていると，そのうち止まったりもします．

11.1.2 while文の意味は?

リスト11.1では6行目に「while (1)」という記述があります．これの意味を説明しましょう．

まずwhile文は，条件を持ちます．これは「(...)」の中身の部分のことです．6行目では条件は「1」のようになっています．

そしてwhile文は，「{ ... }」で括られた「ブロック」を持ちます．7行目の「printf()」が，ブロック内に置かれていることになります．

while文の文法は，if文に似ています．if文は「条件が満たされた場合に，ブロック内の処理を実行する」というものでした．そしてwhile文は，「条件が満たさている間，ブロック内の処理を繰り返し実行する」という動作をします．

リスト11.1では，while文の条件は「1」となっています．これは「常に満たされる」ということになります．

ということは**リスト11.1**は条件が常に満たされるため，ブロック内の処理を永久に続ける，という動作になるわけです．このようなものは「無限ループ」と呼ばれます．

ちなみにwhileの条件が「1」だと「常に満たされる」ですが，「0」にすると「常に満たされない」という意味になります．この場合には，一度も実行されずにループを抜けることになります．

11.1.3 一定の回数だけループさせよう

次にリスト11.2のようなプログラムを作成してみましょう．ファイル名はwhile.cとします．

リスト11.2: while.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(int argc, char *argv[])
005: {
006:         int i;
007:         i = 0;
008:         while (i < 10) {
009:                 printf("i = %d\n", i);
010:                 i = i + 1;
011:         }
012:         printf("i = %d (last)\n", i);
013:         exit(0);
014: }
```

リスト11.2はリスト11.1に比べて，ちょっと複雑になっています．

まず6行目の「int i」は，変数の定義です．ループのカウントに使うための変数として，「i」を定義しています．

さらに7行には「i＝0」という行があります．ここで変数iの値はゼロに初期化されます．

8行目のwhile文は，ループすることを表しています．まず条件がチェックされ，満たされる場合にはブロック内の処理を実行することになります．ブロック内の処理が終わると再び条件のチェックが行われ，満たされた場合には再度ブロック内の処理が実行されます．

つまり「条件チェック」→「ブロック内実行」→「条件チェック」→「ブロック内実行」→「条件チェック」…のように繰り返されるわけです．

9行目と12行目のprintf()は，変数iの値を表示しています．これは，値の確認用です．

さらに10行目に，「i＝i＋1」というものがあります．これについて，ちょっと説明しましょう．

このような式は，「変数iと，iに1を加えた値が等しいというのはおかしい！」のように言われるときがあります．

しかしこれは演算子の意味が異なるためです．C言語では「等しい」という意味の演算子には，9章のif文の説明で出てきた「==」というものが使われます．

そしてC言語では，「=」は「代入する」という意味です．これは数学では「左辺と右辺が等しい」という意味になりますが，C言語では意味が違うのでこの点は注意です．

つまりこれは，「右辺の内容を左辺の変数に代入する」という意味になります．このため例えば「i＝0」と書いたら，「変数iはゼロに等しい」という意味ではなく，「変数iにゼロを代入する」という意味になります．

それを踏まえた上で，もう一度「i＝i＋1」という式を考えてみましょう．

この式の右辺は「i＋1」のようになっています．ということはこれは，変数iに1を加算した値を左辺の変数iに代入する，という意味になります．
例えば変数iの値が2になっている場合には，「i＋1」は「2＋1」で右辺の値は「3」になります．そしてその「3」という値を，左辺の変数iに格納する，ということになります．

つまり「i＝i＋1」という式は，変数iが1だけ増加するという動作になります．

11.1.4 処理の流れを追ってみよう

各行の処理の内容が理解できたところで，**リスト11.2**の処理の流れを追ってみましょう．

順番に見ていくと，以下のような処理をしていることになります．

1．i＝0により，変数iがゼロになる．
2．まず，whileの条件がチェックされる．変数iの値はゼロであり，「i＜10」の条件を満たすのでブロック内の処理に入る
3．ブロックの先頭にはprintf()があるので実行され，変数iの値が表示される
4．ブロックの2行目にはi＝i＋1があるので実行する．変数iの値は1になる．
5．ブロック内の処理がすべて完了したら，whileの条件が再びチェックされる．変数iの値は1であり，「i＜10」の条件を満たすのでブロック内の処理に戻る．
6．ブロック内のprintf()とi＝i＋1が実行される．変数iの値が表示され，値は2になる．
7．whileの条件が再びチェックされる．iの値は2であり，「i＜10」の条件を満たすのでブロック内の処理に戻る．
8．変数iが10になり「i＜10」の条件を満たさなくなるまで，ブロック内の処理が繰り返される．変数iが10になると，whileのブロックの次の行の処理に移る．（whileの処理から抜ける）

while.cをコンパイルして実行すると，以下のようになります．０～９までの値を順番に表示しています．

```
[user@localhost ~]$ gcc while.c -o while
[user@localhost ~]$ ./while
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10 (last)
[user@localhost ~]$
```

最後は変数iが10となったところでループから抜けて，「i = 10 (last)」と出力しています．

これは**リスト11.2**の12行目のprintf()によるものなので，ループから抜けたときの変数iの値は「10」になっていることがわかります．

11.2 for文でのループを試そう

while文によるループを試してみましたが，C言語ではwhile文のほかにもfor文というループの文法があります．

次にfor文によるループを試してみましょう．

11.2.1 for文を使ってみよう

for文の動きを試すために，**リスト11.3**のようなプログラムをfor.cというファイル名で作成します．

なお**リスト11.3**は実は**リスト11.2**と似ているため，while.cをfor.cにコピーして修正することで作成すると，作成が楽かもしれません．

リスト11.3: for.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(int argc, char *argv[])
005: {
006:         int i;
007:         for (i = 0; i < argc; i = i + 1) {
008:                 printf("i = %d\n", i);
009:         }
010:         printf("i = %d (last)\n", i);
011:         exit(0);
012: }
```

for.cをコンパイルして実行してみましょう．実行時には，「a b c」のようにコマンドライン引数を指定します．

```
[user@localhost ~]$ gcc for.c -o for
[user@localhost ~]$ ./for a b c
i = 0
i = 1
i = 2
i = 3
i = 4 (last)
[user@localhost ~]$
```

ループは変数iの値が0～3の範囲で4回繰り返され，最後にループから抜けて「last」と出力されています．

リスト11.3では，実はループは変数 argc の数だけ繰り返しています．変数argcは9章でも説明しましたが，実行時のコマンドライン引数の個数＋1が格納されています．

このため「a b c」のように3個のコマンドライン引数を指定すると，ループは4回繰り返されるわけです．

例えば「a b c d e」のように5個のコマンドライン引数を指定すると，ループは6回繰り返されることになります．やってみましょう．

```
[user@localhost ~]$ ./for a b c d e
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6 (last)
[user@localhost ~]$
```

11.2.2 while文との違いを見てみよう

for.c は while.c と似ていると説明しましたが，どの程度似ているのでしょうか？

ファイルの差分を知りたいときには，8章で説明した diff コマンドが便利です．せっかくなのでここで活用して，while.cとfor.cの差分を見てみましょう．

```
[user@localhost ~]$ diff -u while.c for.c
--- while.c     2014-07-19 09:41:33.814799146 +0900
+++ for.c       2014-07-19 09:41:33.816799149 +0900
@@ -4,10 +4,8 @@
 int main(int argc, char *argv[])
 {
        int i;
-       i = 0;
-       while (i < 10) {
+       for (i = 0; i < argc; i = i + 1) {
                printf("i = %d\n", i);
-               i = i + 1;
        }
        printf("i = %d (last)\n", i);
        exit(0);
[user@localhost ~]$
```

見たところ「i＝0」と「i＝i＋1」の行が無くなり，さらに「while」の行が「for」に変更されています．それ以外の行は，そのままのようです．

11.2.3 while文とfor文を比較してみよう

for文は，while文と似ています．ここで比較してまとめてみましょう．

```
while（ループ条件）{（ブロック）}
for（処理1；ループ条件；処理2）{（ブロック）}
```

「ループ条件」はfor文もwhile文も同じです．まず条件がチェックされ，満たされる場合にはブロック内の処理が実行されることになります．ブロック内の処理の実行後には再度条件がチェックされます．これは条件が満たされるまで，繰り返し行われます．

つまりfor文は，while文に「処理１」と「処理２」が追加されたものということになります．

「処理１」は，初期化処理です．ループの開始前に，一度だけ実行されます．

「処理２」は，ループのブロック内の処理が完了するたびに実行されます．

つまりfor文は，以下の順番で実行されます．

１．「処理１」が実行される
２．「ループ条件」が評価され，条件を満たしていなければループを終了する
３．ブロック内の処理を実行する
４．「処理２」を実行する
５．上記２に戻る

ということはfor文は，以下のようにwhile文を使って書いたものと等価だと言うことができます．

```
処理１
while（ループ条件）{
  （ブロック）
  処理２
}
```

さてここで，もう一度 while.c と for.c の差分（diffコマンドの出力）を見比べてみましょう．

for文の「処理１」は，for.c では「i＝0」のようになっています．また「処理２」は「i＝i＋1」のようになっています．

これは while.c の７行目の処理と，10行目の処理をfor文の処理１，処理２に移動した形になっています．このようにfor文は，while文に「初期化処理」と「ループのたびに実行される処理」を明記できるようにしたもの，と言えます．

なので「処理１」「処理２」には，様々な処理が書けるわけです．

しかしだからといって，何でもかんでも書いてしまうと，それでは余計にわかりにくくなります．このため通常は「そのループ処理に関わる処理」を書くようにします．

ここでは「i＝0」という変数iの初期化と，「i＝i＋1」という変数iのカウント処理が，このループの実行に関わっていると言えます．変数iはループの制御に使われている，「ループ変数」だからです．

なので for.c では，それらを for 文中の「処理１」「処理２」に書くようにしているわけです．

11.3 まとめ

while文とfor文という２つの代表的なループ処理を試すことで，ループ処理というものを見てみました．繰り返し処理は条件分岐と同様に，重要なプログラムの要素になります．

そしてここで気がつくことは，実はwhile文があればfor文と同等のことはできるということです．

またfor文でも，while文の代用はできます．「処理１」と「処理２」の部分を以下のように空欄にしてしまえば，while文と等価になります．

```
for（; ループ条件;）{（ブロック）}
```

なのでこれらはどちらでもいいといえばそのとおりで，処理の内容によって使い分けることになります．一般には何らかの条件が整うまで繰り返すような場合には while，データベースのエントリなどを順番に処理したいような場合にはforが使われることが多いようです．

12

デバッガで動作を追ってみよう
[ツール編5]—GDB

ここまででいくつかのプログラムを作成し，コンパイル・実行して動作を見てきました．

しかし作成したプログラムが，思い通りに動かないこともあるでしょう．そのような場合には「デバッガ」というツールで，プログラムの動作を解析することができます．このような作業はプログラムの誤り（バグ）をとるという意味で，「デバッグ」と呼ばれます．デバッグを補助するツールが「デバッガ」です．

デバッガには様々なものがありますが，広く利用されている「GDB」というデバッガをここでは使ってみます． GDBを利用して，「デバッガ」の操作を体験してみましょう．

12.1 GDBを使ってみよう

GDBはGCCと同様に「GNUプロジェクト」によって開発されている，高機能デバッガです．GCCとの親和性も高く，使いやすいデバッガです．「ジーディービー」と呼ばれます．

ここでは11章で作成したfor.cを，GDBを使って解析してみましょう．

12.1.1 デバッグオプションをつけてコンパイルしよう

デバッグのためには，「デバッグオプション」をつけてプログラムをコンパイルしなおす必要があります．

これには以下のように「-g」というオプションをつけてgccを実行することで，コンパイルしなおします．

```
[user@localhost ~]$ gcc for.c -o for -g
```

これで実行ファイルが作成されます．実行は今まで通りにできます．

```
[user@localhost ~]$ ./for a b c d e
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
```

```
i = 6 (last)
[user@localhost ~]$
```

12.1.2 gdbを起動しよう

さて，gdbを起動してみましょう．

gdbは以下のようにして起動します．ここでは -tui というオプションを付加して起動します．「-tui」を付加することで，ソースコードなどが画面上にスクリーン表示されるようになります．

```
[user@localhost ~]$ gdb -tui for
```

起動すると，**図12.1**のような画面になります．

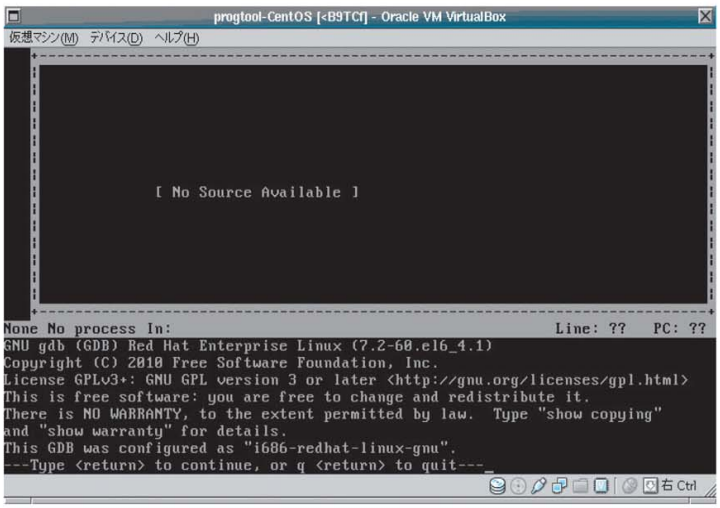


図12.1: gdbを起動する

ライセンス文書が出ていて，一番下には「続行するにはEnterキーを押してください」と出ています．

Enterキーを押すと，次ページの**図12.2**のようになりました．

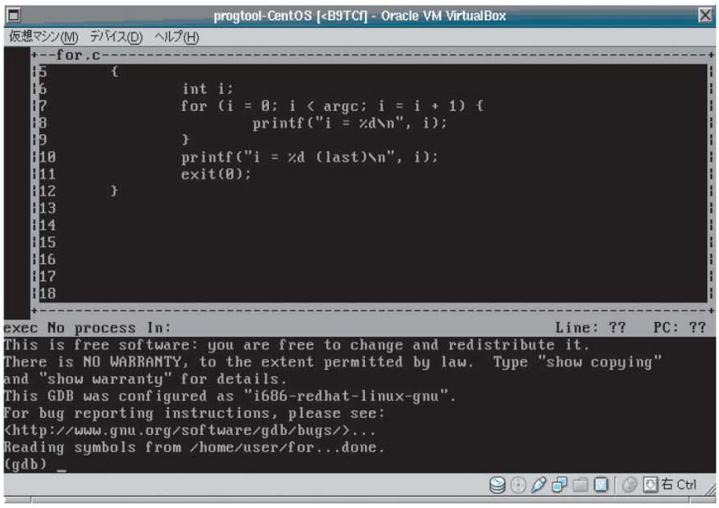


図12.2: Enterを押すとgdbのプロンプトが出る

画面の上半分にはfor.cのソースコードが出ています．さらに一番下の段には，以下のように出ています．

```
(gdb)
```

これはgdbのプロンプトです．ここにコマンド入力することで，gdbを操作することができます．

12.1.3 プログラムを実行してみよう

gdbの操作はコマンドによって行います．まず，単純にプログラムを実行してみましょう．

プログラムは「run」というコマンドで実行できます．コマンドライン引数を「run」の後に続けることもできます． gdbのプロンプトに続けて，以下のように入力してEnterキーを押してみてください．

```
(gdb) run a b c d e
```

これで実行ファイルの「for」が，「a b c d e」というコマンドライン引数で実行されることになります．

実行すると，**図12.3**のような状態になります．

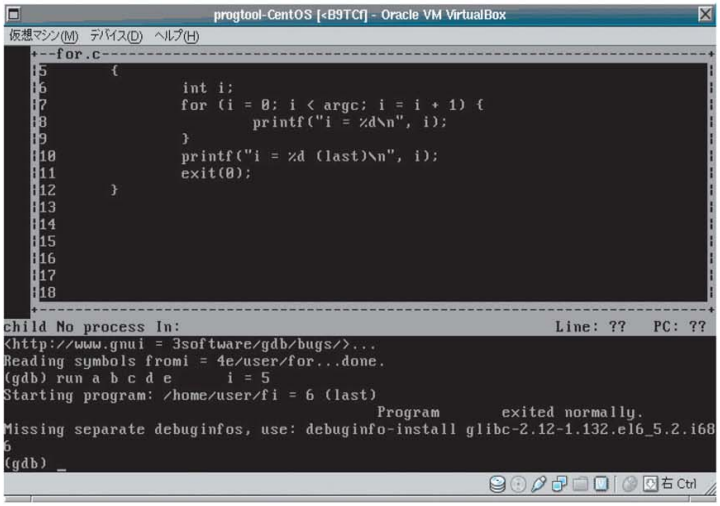


図12.3: プログラムを実行する

実行はできたのですが，残念ながらちょっと画面が崩れてしまっているようです．

どうも -tui オプションを利用すると，このように画面が崩れてしまう場合が多いようです．

-tuiオプションは「Text User Interface」と呼ばれる操作モードを有効にしますが，このモードはCtrl＋X→Aを押すたびに有効／無効を切り替えることができます．

プログラムの出力は，Text User Interfaceを無効にすることで正常に表示されるようになります．出力結果を確認したい場合など，必要に応じてCtrl＋X→AでText User Interfaceを無効化するなど，工夫して使うといいかもしれません．

12.2 デバッガの機能を活用してみよう

これだけだと単に実行できているだけなのですが，デバッガにはデバッグをサポートする機能がたくさんあります．

ここではその中でも代表的な「ブレークポイント」と「ステップ実行」を試してみましょう．

12.2.1 ブレークポイントを設定してみよう

先ほどは単にプログラムを実行しただけでしたが，これでは普通の実行と変わらず，どのように実行されているのかはよくわかりません．

「ブレークポイント」を仕掛けることで，そこでプログラムの実行を一時停止することができます．まずはgdbで以下を実行してみてください．

```
(gdb) break main
```

ここでは「break main」のようにして main() の先頭にブレークポイントを設定しました．

プログラムの実行は main() の先頭から行われるので，つまりプログラムの実行開始直後に，先頭ですぐに一時停止することになります．

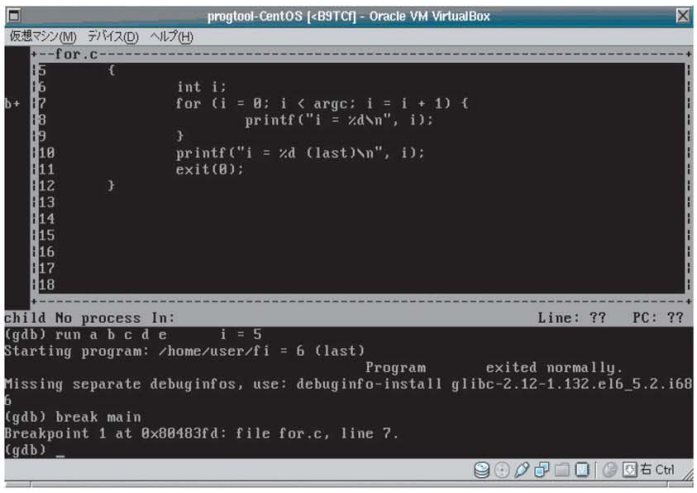


図12.4: ブレークポイントを設定する

「break main」を実行すると，画面は**図12.4**のようになります． for文のある行の左端に「b+」のマークが出ていることがわかります．これが，この行にブレークポイントが張られたという目印です．

for文の行にブレークポイントが仕掛けられたのは，この行がmain()の先頭にあるためです．「先頭にあるのはint iの行では？」と疑問に思うかもしれませんが，「int i」は変数の定義であり実行されるわけではないため，for文の行が先頭となっています．

runで実行してみましょう．

```
(gdb) run a b c d e
```

今度は**図12.5**のような画面になりました．

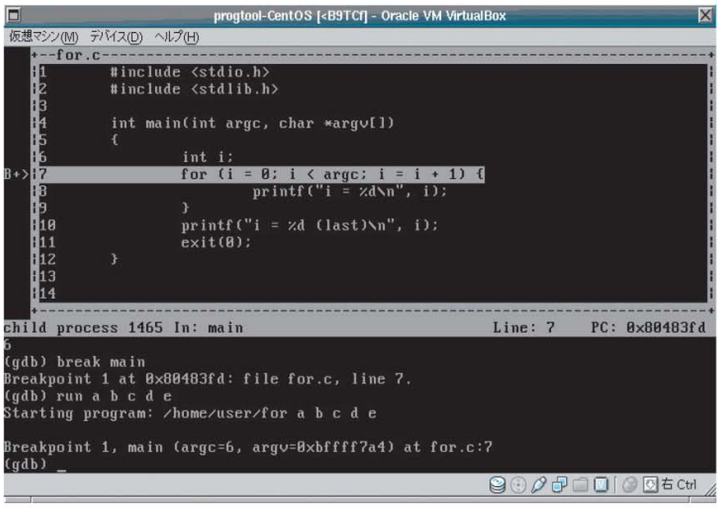


図12.5: ブレークポイントで停止した

main()の先頭にあるfor文の位置が，ハイライトされています．ここが現在のブレーク位置になります．つまりfor文の位置で，実行を一時停止しているわけです．

12.2.2 ステップ実行してみよう

現在は「ブレークポイント」によって実行を一時停止している状態ですが，デバッガには「ブレークポイント」ともうひとつ，便利な機能があります．それは「ステップ実行」です．

「ステップ実行」は，実行を1行ずつ進めるというものです．つまり，すべての行にブレークポイントが仕掛けられている状態で実行を進めるような感じです．
ブレークポイントで一時停止した状態から「next」と入力することで，ステップ実行します．

```
(gdb) next
```

画面は**図12.6**のようになりました． for文の次にあるprintf()の行がハイライトされ，for文の行から実行が次に進んだことがわかります．

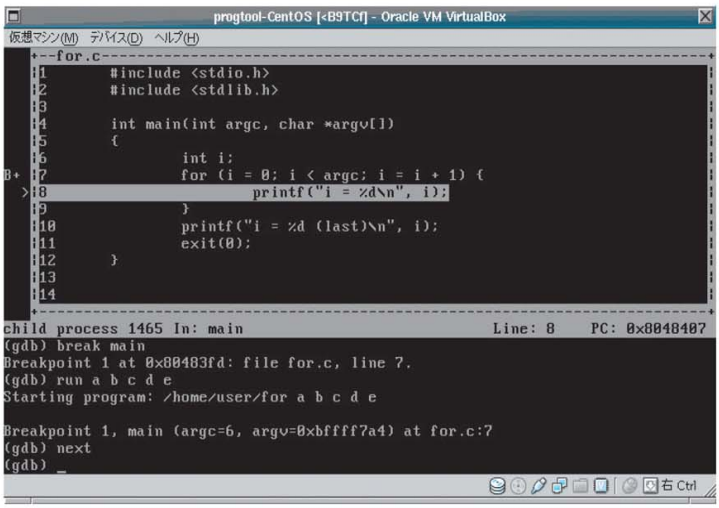


図12.6: ステップ実行する

もう一度，ステップ実行をしてみます．

```
(gdb) next
```

今度は画面は**図12.7**のようになりました．printf()の行の実行が完了し，またfor文の行に実行が戻っています．

これはまるで実行が逆戻りしているようにも見えますが，そうではありません．for文によるループが行われ，実行が進んだためにこのようになっています．つまりループの２周目に入っているわけです．

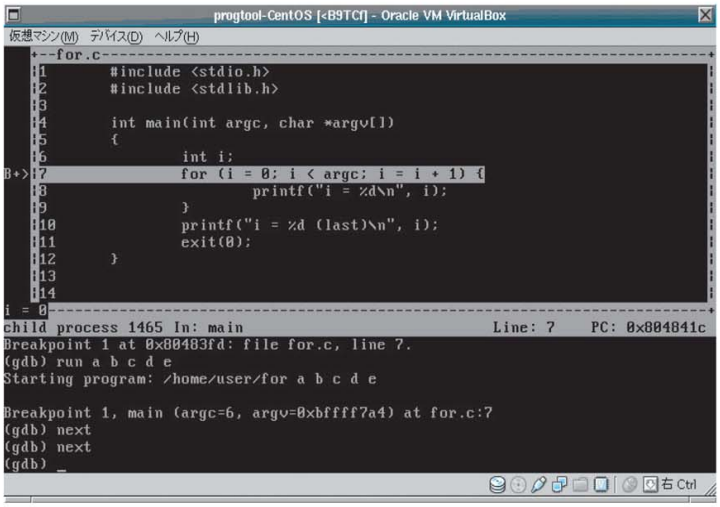


図12.7: ステップ実行を進める

そして本来ならば，ここでprintf()が実行されて「i＝0」と表示されているはずです．

しかし残念ながら -tui を付加していると，このような表示がうまく行われないようです．Ctrl＋X → A で Text User Interface の有効／無効を切り替えながらステップ実行を進めてみるといいかもしれません．

12.2.3 変数の値を表示してみよう

一時停止している状態で，実行中の変数の値を知ることもできます．

以下を実行してみましょう．

```
(gdb) print i
```

これで**図12.8**のように，変数iの値が表示されます．

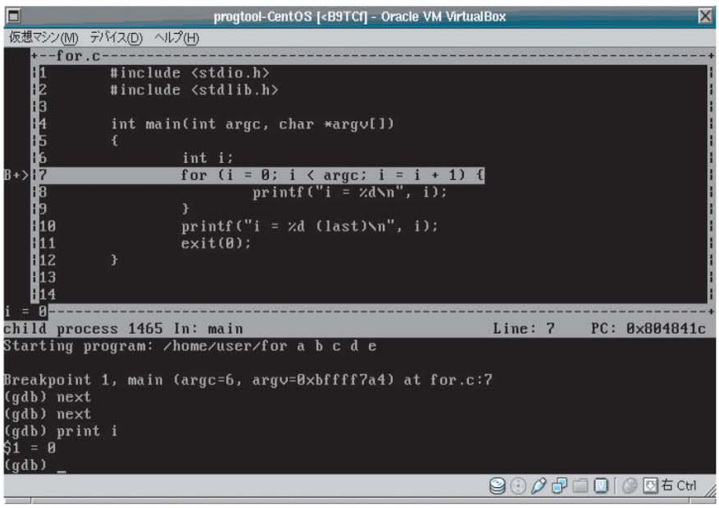


図12.8: 変数iの値を表示する

「$1＝0」と表示されています．これが変数iの値です．

今，ハイライトはforの行にあります．実行はループの1周目が終わり，forの先頭に戻っています．

ただしハイライトされている行は，「実行が終わった行」ではなく「これから実行される行」です．このためfor文中の「i＝i＋1」はまだ実行されておらず，変数iの値は初期値のゼロになっているわけです．

nextでステップ実行をひとつ進め，もう一度printで変数iの値を表示させてみましょう．

```
(gdb) next
(gdb) print i
```

結果は**図12.9**のようになりました．ハイライトがprintf()の行に移動し，変数iの値は今度は「1」と表示されました．

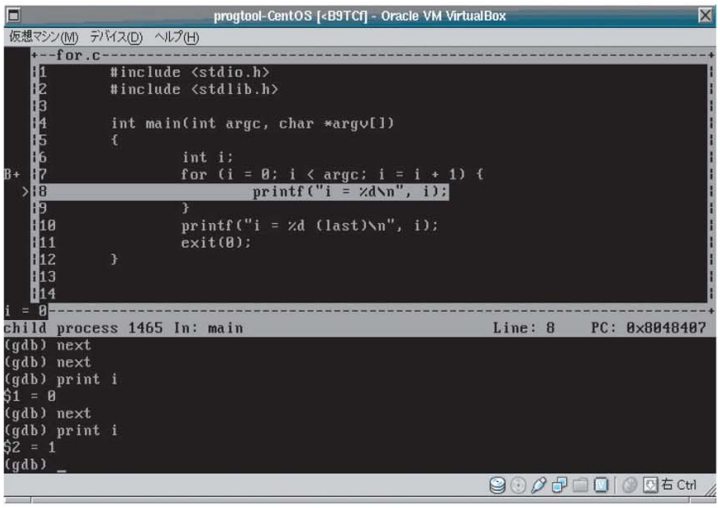


図12.9: 変数iを再度表示する

12.2.4 ステップ実行を繰り返してループから抜けよう

さて，実行をさらに進めてみましょう．

nextを繰り返していけば，いずれ変数iの値が変数argcと等しくなり，ループから抜けることができるはずです．

図12.10は何度かnextの実行を繰り返し，ちょうどループから抜け出たところです．

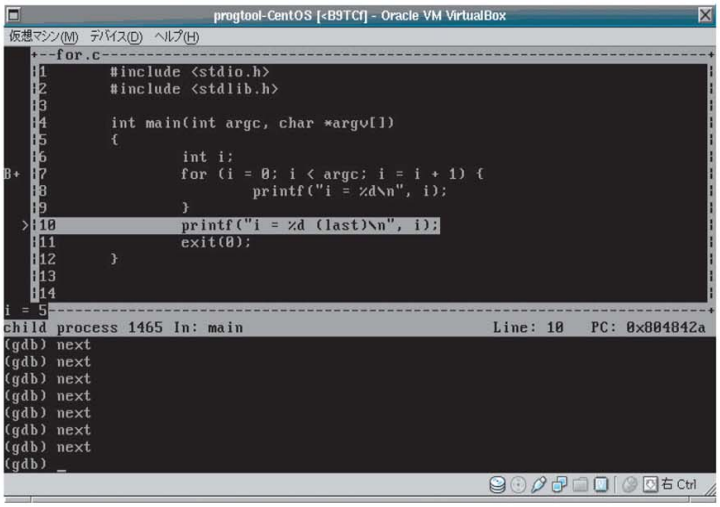


図12.10: ステップ実行を繰り返してループから抜ける

図12.10ではループを抜けたため，ループ直後のprintf()の行がハイライトされています．

この状態でもう一度，変数iの値を見てみましょう．

```
(gdb) print i
```

実行結果は**図12.11**のようになりました．今度は「6」になっています．

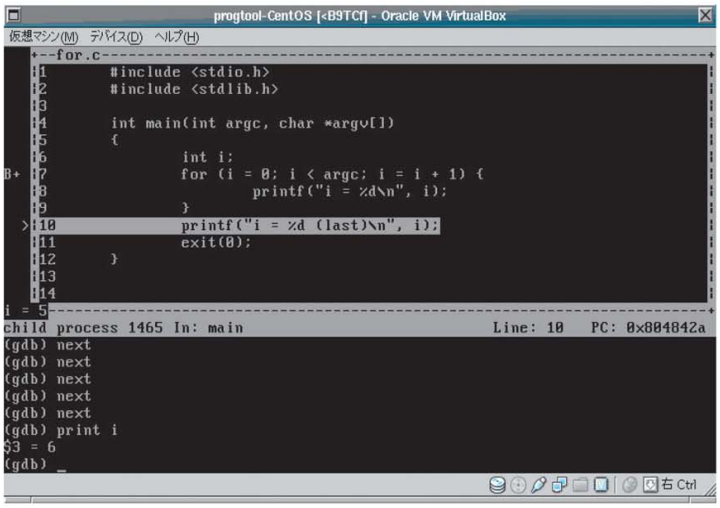


図12.11: 変数iが６になっている

変数argcの値はいくつだったのでしょうか．「print argc」で確認してみましょう．

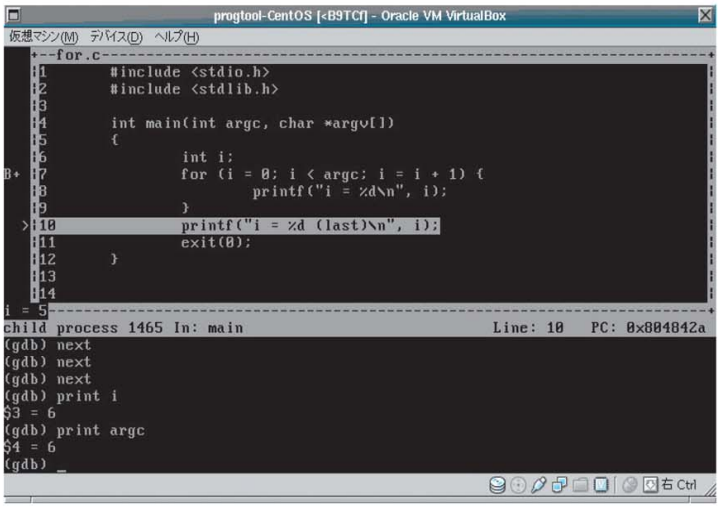


図12.12: 変数argcも６になっている

変数argcの値も「６」です．

実行時の引数は「ａｂｃｄｅ」なので５個です．変数argcにはコマンドライン引数の数＋１の値が渡されますから，これは納得がいきます．

そしてfor文のループ条件は「i < argc」です．つまり変数iは，変数argcと同じ値になったことでループ条件を満たさなくなり，ループから抜けたことになります．

12.2.5 実行を継続してみよう

最後に「continue」を実行すると，一時停止を解除して実行を再開します．

```
(gdb) continue
```

continueを実行すると，**図12.13**のようになりました．「Program exited normally.」と表示されているので，プログラムは正常終了したようです．

つまり実行を再開し，そのままプログラム終了まで一気に到達したようです．

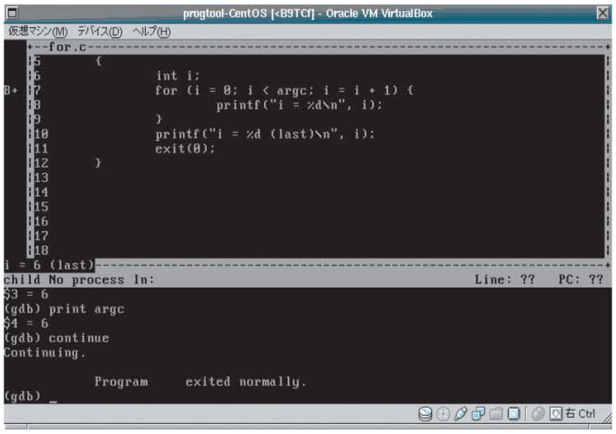


図12.13: continueで実行を再開する

最後に，gdbは「quit」を実行することで終了します．

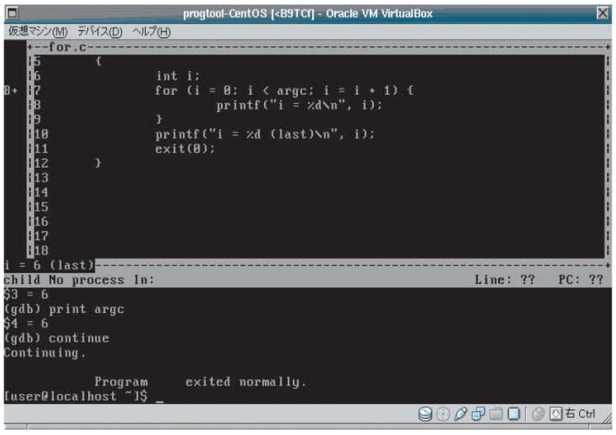


図12.14: quitで終了する

12.3 まとめ

プログラム開発では，デバッガの利用は必須と言ってもいいでしょう．デバッガは自身のプログラムのデバッグにも役立ちますが，与えられたプログラムの動作を理解したり，解析したりするときにも重宝します．

またデバッガはGUIのものも多くあります．しかしそれらはGDBにGUIのインターフェースをかぶせたものも多く，裏ではGDBが動作しているということも少なくありません．

そうしたツールでは，いざというときにはCUIでの操作もできるインターフェースが提供してあることもあります．このためCUIによるGDBの操作は，一度触れておいても損は無いといえるでしょう．

●本書で紹介するCentOSのCUIコマンドー3（4・5章）

コマンド		ツールの役割	ページ
nano（ファイル名）		（ファイル名）をテキストエディタnanoで開く	100,110
	^G「Ctrl」+「G」	ヘルプ画面表示	105
	^X「Ctrl」+「X」	画面から抜ける/nanoの終了	105
	^K「Ctrl」+「K」	カット	106
	^U「Ctrl」+「U」	ペースト	106
vi（ファイル名）		（ファイル名）をテキストエディタviで開く	115
	I	入力モードに入る	116
	[Esc]キー	コマンドモードに入る	119
	X	カーソルの位置の文字を1文字削除	119
	DD	現在の行を削除する	120
	P	削除したテキストをペーストする	120
	:	コマンド入力	121
	W	ファイル書き込み	122
	Q	終了	122
gcc（Cプログラムファイル名） -o（実行ファイル名）		GCCで（Cプログラムファイル名）をコンパイルする	127
./（実行ファイル名）		カレントディレクトリにある（実行ファイル名）を実行する	130
objdump -d（実行ファイル名）> （結果ファイル）.dis		実行ファイルを逆アセンブルし（結果ファイル）.disに出力する	143

13

アルゴリズムを考えてみよう

[C言語プログラミング編5]—配列

デバッガを体験した後には，またC言語のプログラミングに戻ります．

ここまでで「条件分岐」と「ループ」という構文を使ってみました．これらはプログラムの基本構造となるため，実はこれらをマスターすれば，いろいろなプログラムを書くことができるようになります．

本章ではその例として，定番の「ソート」のプログラムを書いてみます．そしてコンピュータの処理手順である「アルゴリズム」を考えてみましょう．

13.1 「ソート」のアルゴリズムを考えてみよう

「ソート」とは，並びかえのことです．

例えば以下のような数字の列があるとしましょう．

```
5 3 4 2 1
```

これは１から５までの数字を適当に並べたものです．これを，小さい順に並びかえてみましょう．

```
1 2 3 4 5
```

このような並びかえ作業のことを「ソート」と呼びます．

ソートは「数字の大きい順」や「辞書順」など，いくつかの方法が考えられます．また「大きい順」「小さい順」といった順番も考えられるでしょう．

しかしここでは単純に，「数字の小さい順」の並び替えを考えてみます．

13.1.1 コンピュータの制限とは？

１から５までの数を並びかえるとき，それが我々人間なら，どのように作業を行うでしょうか？

もう一度，数字の列を見直してみましょう．

```
5 3 4 2 1
```

これくらいの数の量ならば，パッと見た感じで並べかえてしまうでしょう．

しかしコンピュータはそうはいきません．人間とコンピュータには，以下のような違いがあります．

- **約束１「コンピュータは，同時に２つの数しか比較できない」**
  これはif文を思い浮かべてください．コンピュータは「全体を見て一番小さな値を選ぶ」といったことはできず，比較できるのはつねに「２つの数」です．
- **約束２「変数は，一度にひとつだけ上書きできる」**
  これは変数の操作を思い浮かべてください．変数は「a＝1」「b＝2」のようにして，一度にひとつの変数に対して代入できます．値の入れ替えなどはできません．
- **約束３「数は，変数という置き場所にしか置けない」**
  数値を扱う場合には，必ず「変数」を使う必要があります．「どこかそのへんの適当なところに置いておく」ということはできません．
- **約束４「手順は，すべて決めておかなければならない」**
  並べ替えの手順は，「手順書」として最初に決めておく必要があります．「最初は一番小さな数を適当に選んで，最後のほうは整列させるように臨機応変に対応する」のような，曖昧なことはできません．

この条件を課した上で，上の数字の列を並びかえることを考えてみましょう．

```
5 3 4 2 1
```

まずこれらの５つの数が，a,b,c,d,eの変数に，順番にそれぞれ格納されているとします．つまり変数aには先頭の「５」が，変数bには次の「３」が…という具合です．

「約束１」を見てください．コンピュータは同時に２つの数の比較しか行えないため，全体的に見て一気に比較するようなことはできません．

ということで，まず先頭の２つの数を比較しましょう．つまり変数aとbを比較するわけです．

変数aとbの値は「５」と「３」ですから，「小さい順」にすることを考えると，順番は逆になっています．このため，これらの値をひとまず交換すれば，ソート作業は部分的に進むことになります．

しかし値の交換にも，ひと工夫が必要になります．

13.1.2 値の入れ替えをするには

数字を交換するときのことを考えてみます．

人間がやるならば，両手で5と3を持って入れ替えてしまうでしょう．しかし「約束2」を見てください．コンピュータは，一度にひとつの数しか扱えないのです．

これは，片手しか使ってはいけないというイメージです．このような場合，どうやって入れ替えればいいでしょうか？

次に思いつくのは，「3」という数字をまずつかんでどこかに置いておき，「5」を変数bに移動してから，「3」を変数aに移動する，という手順です．

しかしこれもダメです．「約束3」を思い出してください．数字はそのへんに置いておく，ということはできません．

ということは，交換のための値の一時的な置き場所として，別の「変数」を用意する必要があるということです．

ここではその置き場所として「変数x」を作成して利用することにしましょう．つまり値の入れ替えは，以下のような手順になります．これならば，片手で行うことも可能でしょう．

1．変数bから「3」をつかんで変数xに移動する．
2．変数aから「5」をつかんで変数bに移動する．
3．変数xから「3」をつかんで変数aに移動する．

これで，変数aと変数bの値を入れ替えることができます．これはC言語で書くと，以下のような処理になるでしょう．

```
x = b;
b = a;
a = x;
```

13.1.3 並べかえのアルゴリズムを考えよう

さて，値の比較と入れ替えはできるようになりました．

これらの作業を順次繰り返していけば徐々にソートが進んで，そのうち変数a～eは並びかえられることになります．以下のような手順で並びかえていけばいいでしょうか．

1．aとbを比較し，順番が逆ならば入れ替える
2．bとcを比較し，順番が逆ならば入れ替える
3．cとdを比較し，順番が逆ならば入れ替える
4．dとeを比較し，順番が逆ならば入れ替える
5．1～4までを繰り返す

最後に「約束4」を思い出してください．並べ替えの手順は，このように手順書として最初に決めておく必要があります．つまりこれが「プログラム」です．

そしてこのような手順のことを「アルゴリズム」と呼びます．つまりこれは「ソートのアルゴリズム」です．

13.1.4 プログラムにしてみよう

この「アルゴリズム」をC言語のプログラムにしてみましょう．値の入れ替えは変数xを使うことで可能です．また繰り返しは，11章でつかった「ループ」によって実現できます．

すると**リスト13.1**のようなプログラムになりました．

リスト13.1: sort.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(int argc, char *argv[])
005: {
006:         int a, b, c, d, e;
007:         int x;
008:
009:         a = 5;
010:         b = 3;
011:         c = 4;
```

```
012:          d = 2;
013:          e = 1;
014:
015:          while (1) {
016:                  printf("%d %d %d %d %d\n", a, b, c, d, e);
017:                  if (a > b) {
018:                          x = b;
019:                          b = a;
020:                          a = x;
021:                  }
022:                  if (b > c) {
023:                          x = c;
024:                          c = b;
025:                          b = x;
026:                  }
027:                  if (c > d) {
028:                          x = d;
029:                          d = c;
030:                          c = x;
031:                  }
032:                  if (d > e) {
033:                          x = e;
034:                          e = d;
035:                          d = x;
036:                  }
037:          }
038:          exit(0);
039: }
```

今まで書いたプログラムと比べてちょっと長めですが，これは変数a～eの処理をしているためであり，似た部分が多いため，難しいことは無いと思います．

リスト13.1を実行してみましょう.

```
[user@localhost ~]$ gcc sort.c -o sort
[user@localhost ~]$ ./sort
5 3 4 2 1
3 4 2 1 5
3 2 1 4 5
2 1 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
... (無限ループになるので，Ctrl+Cで中断する) ...
1 2 3 4 5
1 2 3 4 5
^C[user@localhost ~]$
```

実行すると，「1 2 3 4 5」という行が延々と表示されて，実際には上のほうのソート経過はすぐに画面外に流れていって見えなくなってしまいます．無限ループに陥ってあせってしまうかもしれませんが，11章で説明した「Ctrl＋C」で中断してください.

これは**リスト13.1**では並べ替えを行ってはいるのですが，並べ替えが終わったことを検知せず，ループを継続しているためです．つまり「1 2 3 4 5」のように並べ替えられてこれ以上やることが無くなっても，並べ替え作業を無限に続けてしまっているわけです.

このような場合には，3章で説明した「headコマンド」が利用できます．以下のようにして先頭のみ表示させてみましょう.

```
[user@localhost ~]$ ./sort | head -n 5
5 3 4 2 1
3 4 2 1 5
3 2 1 4 5
2 1 3 4 5
1 2 3 4 5
[user@localhost ~]$
```

最終的には「1 2 3 4 5」のようになっていますから，ソートが期待通りに行われているようです.

もしも実行結果が一致しない場合には，変数a～eの扱いに間違いが無いか，sort.cを細かくチェックしてみてください.

13.2 完了を検出できるようにしよう

とりあえずソートはできているのですが，並べかえが終わっても延々と繰り返してしまっています．

並べ替えの完了を検知して終了するように，改良してみましょう．

13.2.1 gitでソースコードを管理しよう

改良の前に，せっかくなのでgitでソースコード管理してみましょう．

gitリポジトリは10章で作成していますので，それを流用します．もしも削除してしまった場合には，mkdirでディレクトリを作成しディレクトリ内で「git init」を実行することで，リポジトリを作成してください．

まず「sort」というディレクトリをgitリポジトリ内に作成します．

```
[user@localhost ~]$ cd git
[user@localhost git]$ mkdir sort
[user@localhost git]$ cd sort
[user@localhost sort]$
```

そしてそこに**リスト13.1**のsort.cを登録しましょう．

```
[user@localhost sort]$ mv ../../sort.c .
[user@localhost sort]$ git add sort.c
[user@localhost sort]$ git commit sort.c
```

コミット時のコメントは適当に入力しておきます．

これでsort.cが登録できましたので，履歴を確認しておきましょう．

```
[user@localhost sort]$ git log sort.c
commit 655b64d6a76f950a512997f29abdf10efbb4f495
Author: SAKAI Hiroaki
Date:   Sat Jul 12 18:54:52 2014 +0900

    Sort sample program.
[user@localhost sort]$
```

13.2.2 完了を検出する方法を考えよう

gitリポジトリにファイルを登録した後はいつでも元に戻せますから，安心して改造していくことができます．

まずは並べ替えの完了を検出して終了するように改造したいわけですが，これはどのようにすれば検出できるでしょうか？ちょっと考えてみましょう．

ソートがされている最中は，変数の値の入れ替えが発生しています．ということは変数の比較をひととおり行ったときに，変数の値の入れ替えが一度も発生しなければ，並べ替えは完了していると言うことができそうです．

つまり「入れ替えが起きたかどうか」という情報を保存しておけばいいわけです．コンピュータでの保存の場所は，やはり「変数」です．つまり「入れ替えが起きた」という状態も，変数を使って保持することができます．

これを「changed」という変数で保持し，入れ替えが無かった (つまり，changedがゼロのままだった)ときに終了するようにしてみましょう．このような変数を「フラグ」と呼びます． sort.c を**リスト13.2**のように変更し，ひとまずフラグ changed を追加してみます．

リスト13.2: フラグを追加する

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(int argc, char *argv[])
005: {
006:         int a, b, c, d, e;
007:         int x;
008:         int changed;
009:
010:         a = 5;
011:         b = 3;
012:         c = 4;
013:         d = 2;
014:         e = 1;
015:
016:         changed = 1;
017:         while (changed == 1) {
018:                 printf("%d %d %d %d %d\n", a, b, c, d, e);
019:                 if (a > b) {
020:                         x = b;
```

```
021:                            b = a;
022:                            a = x;
023:                    }
024:                    if (b > c) {
025:                            x = c;
026:                            c = b;
027:                            b = x;
028:                    }
029:                    if (c > d) {
030:                            x = d;
031:                            d = c;
032:                            c = x;
033:                    }
034:                    if (d > e) {
035:                            x = e;
036:                            e = d;
037:                            d = x;
038:                    }
039:            }
040:            exit(0);
041: }
```

git diff で sort.c の変更の差分を確認してみましょう.

```
[user@localhost sort]$ git diff sort.c
diff --git a/sort/sort.c b/sort/sort.c
index eea4212..12355fa 100644
--- a/sort/sort.c
+++ b/sort/sort.c
@@ -5,6 +5,7 @@ int main(int argc, char *argv[])
 {
        int a, b, c, d, e;
        int x;
+       int changed;

        a = 5;
        b = 3;
@@ -12,7 +13,8 @@ int main(int argc, char *argv[])
        d = 2;
        e = 1;

-       while (1) {
+       changed = 1;
+       while (changed == 1) {
                printf("%d %d %d %d %d\n", a, b, c, d, e);
                if (a > b) {
                        x = b;
[user@localhost sort]$
```

changed の処理が追加されていることがわかります．ただし changed が 1 以外のときに終了するようにしただけで，実際に changed を 1 以外に設定している部分は無いため，これは changed により処理を終了する「テンプレート」を追加しただけと言えます．

ひとまず変更をコミットしておきましょう．このように部分的に修正して細かくコミットすることで，後々に変更履歴がわかりやすくなります．

```
[user@localhost sort]$ git add sort.c
[user@localhost sort]$ git commit sort.c
```

13.2.3 フラグを適切に設定しよう

この段階では，終了するための判定用のフラグを追加しただけで，フラグを状況に応じて値を設定してはいません．このため**リスト 13.2** は，やはり終了を検知できません．

状況に応じて changed を再設定するようにしてみましょう．以下のような処理を追加すればよさそうです．

1. **比較処理の先頭で，変数 changed をゼロにする**
2. **変数 a と b を比較し，順番が逆ならば入れ替え後に変数 changed を 1 にする**
3. **変数 b と c を比較し，順番が逆ならば入れ替え後に変数 changed を 1 にする**
4. **変数 c と d を比較し，順番が逆ならば入れ替え後に変数 changed を 1 にする**
5. **変数 d と e を比較し，順番が逆ならば入れ替え後に変数 changed を 1 にする**

このようにすれば，変更があった場合に changed が 1 となり，変更が無ければ changed はゼロのままということになります．

実際に処理を追加すると，**リスト 13.3** のようになりました．

リスト 13.3: フラグを設定する

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(int argc, char *argv[])
005: {
006:         int a, b, c, d, e;
007:         int x;
008:         int changed;
```

```
009:
010:          a = 5;
011:          b = 3;
012:          c = 4;
013:          d = 2;
014:          e = 1;
015:
016:          changed = 1;
017:          while (changed == 1) {
018:                  printf("%d %d %d %d %d\n", a, b, c, d, e);
019:                  changed = 0;
020:                  if (a > b) {
021:                          x = b;
022:                          b = a;
023:                          a = x;
024:                          changed = 1;
025:                  }
026:                  if (b > c) {
027:                          x = c;
028:                          c = b;
029:                          b = x;
030:                          changed = 1;
031:                  }
032:                  if (c > d) {
033:                          x = d;
034:                          d = c;
035:                          c = x;
036:                          changed = 1;
037:                  }
038:                  if (d > e) {
039:                          x = e;
040:                          e = d;
041:                          d = x;
042:                          changed = 1;
043:                  }
044:          }
045:          exit(0);
046: }
```

変更の差分を確認しましょう．

```
[user@localhost sort]$ git diff sort.c
```

git diff を実行すると，**図13.1**のようにビューワが起動します．これは修正内容が1画面内に収まりきれないためです．矢印キーの「↑」「↓」で上下に移動し，修正内容の全体を見ることができます．ビューワは「Q」を押すことで終了します．

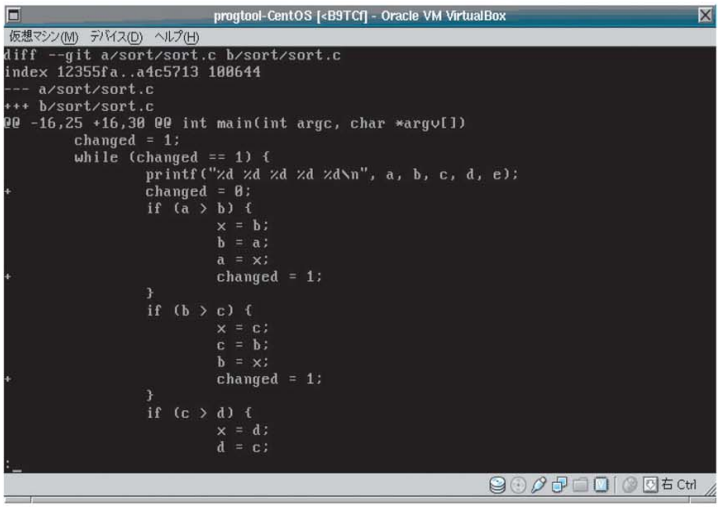


図 13.1: git diff の実行画面

修正内容の全体は，以下のようになります．

```
[user@localhost sort]$ git diff sort.c
diff --git a/sort/sort.c b/sort/sort.c
index 12355fa..a4c5713 100644
--- a/sort/sort.c
+++ b/sort/sort.c
@@ -16,25 +16,30 @@ int main(int argc, char *argv[])
        changed = 1;
        while (changed == 1) {
                printf("%d %d %d %d %d\n", a, b, c, d, e);
+               changed = 0;
                if (a > b) {
                        x = b;
                        b = a;
                        a = x;
+                       changed = 1;
                }
                if (b > c) {
                        x = c;
                        c = b;
                        b = x;
+                       changed = 1;
                }
                if (c > d) {
                        x = d;
                        d = c;
                        c = x;
+                       changed = 1;
```

```
                }
                if (d > e) {
                        x = e;
                        e = d;
                        d = x;
+                       changed = 1;
                }
        }
        exit(0);
[user@localhost sort]$
```

修正内容を見ると，変数changedのゼロクリア処理と，各々の比較場所での変数changedの更新が追加されていることがわかるでしょう．

実行してみましょう．

```
[user@localhost sort]$ gcc sort.c -o sort
[user@localhost sort]$ ./sort
5 3 4 2 1
3 4 2 1 5
3 2 1 4 5
2 1 3 4 5
1 2 3 4 5
[user@localhost sort]$
```

今度はソート完了後に，プログラムが終了しています．終了を検出できているようです．

うまく動いたので，git commit をして登録しておきましょう．

```
[user@localhost sort]$ git add sort.c
[user@localhost sort]$ git commit sort.c
```

13.2.4 配列を使ってみよう

これでも並べ替えはできるのですが，**リスト13.3**には似たような処理の部分があります．各変数の比較部分です．これはちょっと冗長なように思えます．

この部分は「繰り返し」によって，もっとシンプルに表現できそうです．

そこで「繰り返し」を使うように変更して…と言いたいところですが，その前にまずは「配列」というものを使って sort.c を書き換えてみます．

変数はa,b,c,d,eの５種類を利用しています．まずはこれらを番号づけしてみましょう．C言語では「配列」と言って，「a[0]」や「a[1]」のような番号つきの変数を利用することができます．

例えばa～eのかわりに，a[0],a[1],a[2],a[3],a[4]という５つの変数を利用してsort.cを書き換えてみましょう．

リスト13.4: 配列を利用する

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(int argc, char *argv[])
005: {
006:         int a[5] = { 5, 3, 4, 2, 1 };
007:         int x;
008:         int changed;
009:
010:         changed = 1;
011:         while (changed == 1) {
012:                 printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
013:                 changed = 0;
014:                 if (a[0] > a[1]) {
015:                         x = a[1];
016:                         a[1] = a[0];
017:                         a[0] = x;
018:                         changed = 1;
019:                 }
020:                 if (a[1] > a[2]) {
021:                         x = a[2];
022:                         a[2] = a[1];
023:                         a[1] = x;
024:                         changed = 1;
025:                 }
026:                 if (a[2] > a[3]) {
027:                         x = a[3];
028:                         a[3] = a[2];
029:                         a[2] = x;
030:                         changed = 1;
031:                 }
032:                 if (a[3] > a[4]) {
033:                         x = a[4];
034:                         a[4] = a[3];
035:                         a[3] = x;
036:                         changed = 1;
037:                 }
038:         }
039:         exit(0);
040: }
```

リスト13.4は変数a→a[0]，変数b→a[1]，変数c→a[2]，... のように，単純に置き換えただけです．また配列を利用することで，変数の初期化は以下のように1行でまとめて書くことができるようになりました．

```
        int a[5] = { 5, 3, 4, 2, 1 };
```

git diff で sort.c の変更点を見てみましょう．**図13.2**は git diff sort.c の実行後，ビューワが起動したら矢印キーの「↓」で少し下に移動したところです．変更箇所が多数あることがわかるでしょう．これは変数の置き換えを行っているので，変更行が全体的に発生してしまっているためです．

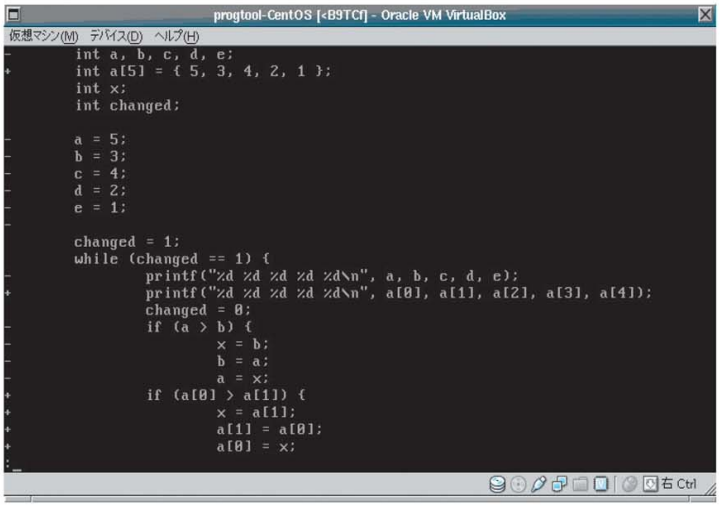


図13.2: 変更箇所が多数存在する

sort.c をコンパイルして実行してみましょう．

```
[user@localhost sort]$ gcc sort.c -o sort
[user@localhost sort]$ ./sort
5 3 4 2 1
3 4 2 1 5
3 2 1 4 5
2 1 3 4 5
1 2 3 4 5
[user@localhost sort]$
```

問題無く実行できています．動作に変わりは無いようです．

これもコミットしておきましょう．

```
[user@localhost sort]$ git add sort.c
[user@localhost sort]$ git commit sort.c
```

13.2.5 配列とループと組み合わせよう

配列の便利なところは，a[0]やa[1]のようにして，番号を使って変数を指定できることです．これはa[i]のように，変数で番号指定することもできます．このためループで各変数に対してひとつずつ処理をしていく，という書き方ができます．

例えば5つの変数の値をすべてゼロに設定する場合のことを考えてみましょう．これを変数a～eに対して行うならば，以下のように書く必要があります．

```
a = 0;
b = 0;
c = 0;
d = 0;
e = 0;
```

しかしこれは，配列を使えば以下のようにループを利用して書くことができます．

```
for (i = 0; i < 5; i++) {
  a[i] = 0;
}
```

上の例だと変数iが0～4まで順番にカウントされます．このためまず「a[0]＝0」が行われ，2周目は「a[1]＝0」が，3周目は「a[2]＝0」がというように，順に変数がゼロに設定されるわけです．

このように配列はループと相性がいいのですが，するとソートの処理は以下のように書き換えることができます．

1．変数changedをゼロにする
2．変数iを使って，0～3までループする
3．a[i]とa[i+1]を比較し，順番が逆ならば入れ替えて変数changedを1にする
4．2～3の処理をループする
5．変数changedが1ならば終了する．ゼロならば1に戻る

これはC言語で書くと，**リスト13.5**のようになります．

リスト13.5: ループ処理で置き換える

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(int argc, char *argv[])
005: {
006:         int a[5] = { 5, 3, 4, 2, 1 };
007:         int x;
008:         int changed;
009:         int i;
010:
011:         changed = 1;
012:         while (changed == 1) {
013:                 printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
014:                 changed = 0;
015:                 for (i = 0; i < 4; i++) {
016:                         if (a[i] > a[i + 1]) {
017:                                 x = a[i + 1];
018:                                 a[i + 1] = a[i];
019:                                 a[i] = x;
020:                                 changed = 1;
021:                         }
022:                 }
023:         }
024:         exit(0);
025: }
```

リスト13.4に比べて，ずいぶん短くすることができました．

実行してみましょう．

```
[user@localhost sort]$ gcc sort.c -o sort
[user@localhost sort]$ ./sort
5 3 4 2 1
3 4 2 1 5
3 2 1 4 5
2 1 3 4 5
1 2 3 4 5
[user@localhost sort]$
```

問題なく実行できています．つまり配列とループを利用することで，プログラムを等価でさらにシンプルなものにすることができたわけです．

最後に**リスト13.5**を，ソートプログラムの完成形として登録しておきましょう．

```
[user@localhost sort]$ git add sort.c
[user@localhost sort]$ git commit sort.c
```

13.3 まとめ

アルゴリズムは処理の手順のことですが，結果は同じでも，手順は何種類かが存在するのが普通です．

例えばここで紹介したソートのアルゴリズムは「バブルソート」と呼ばれるものです．
ソートの経過を見てみると，数字がまるで泡のように先頭に向かって上がっていくような動きをしていて，そこから来ている呼び名です．

しかしソートには他にも「マージソート」「クイックソート」などのアルゴリズムがあり，速度効率が異なってきます．これは何万，何十万というデータをソートしようとするときに効いてきます．バブルソートは実行時間がデータ数の２乗に比例して増加するため，データ量が増えると実行時間が極端に大きくなってしまいます．これを「スケーラビリティが無い」と言います．

しかしバブルソートもメリットはあります．それは「ワーク領域を必要としない」という点です．ワーク領域というのは，作業のために一時的に必要となるような変数領域のことです．厳密にいうとsort.cでは値の交換のために変数xを利用していますがこれは入れ替えのためであり，ソートの処理自体にはワーク領域を必要としていません．つまり，データ量に応じたサイズのワーク領域を必要としているようなことは無く，データ量が増えれば必要なワーク領域のサイズも増加する，ということもありません．これは「速度的なスケーラビリティは低いがメモリ容量的にはスケーラビリティがある」ということです．

このためアルゴリズムは「使い分け」が重要になります．データ量が大きくなり速度が重視されるときには，速度的なスケーラビリティのあるアルゴリズムを選択するが，データ量が限られていてむしろメモリ量を節約したいような場合には，バブルソートのようなアルゴリズムを使う，というように選択できることが大切です．

14

コンパイルを自動化してみよう
[ツール編6]—makeコマンド

ここまでC言語のプログラムを実行するためには，コンパイルして実行ファイルを作成していました．そしてコンパイルのためには，gccというコマンドを実行していました．

しかし，毎回これを実行するのは面倒です．

定型の処理は自動化してしまいましょう．そしてこのような自動化は，「make」というコマンドを利用することで実現することができます．

14.1 makeコマンドを使ってみよう

makeコマンドを利用するためには，その「手順書」である Makefile というファイルを作成する必要があります．makeは「メイク」，Makefileは「メイクファイル」と読みます．

まず手始めに，Makefile を作成してみましょう．

14.1.1 Makefileを作成してみよう

ここでは前章で作成した sort.c の完成形をコンパイル用のサンプルとして使いたいため，まずは sort.c を登録した git リポジトリに入ります．

git リポジトリが無ければ，sort.c が置いてあるどこかのディレクトリでも構いませんし，sort.c を新たに適当に作成しても構いません．

```
[user@localhost ~]$ cd git/sort
[user@localhost sort]$
```

ここで「Makefile」というファイル名で，**リスト14.1**のようなファイルを作成します．

注意として，「gcc」と「rm」の前にある８文字ぶんの空白は，空白文字ではなくタブ文字（「Tab」というキーを押すことで入力する）１個とします．

Makefileではタブ文字と空白は違うものとして区別されるので注意してください．また余分な空白なども無いようにしてください．

リスト14.1: Makefile

```
001: all :
002:         gcc sort.c -o sort
003:
004: clean :
005:         rm -f sort
```

14.1.2 makeを実行してみよう

ファイルを作成したら，まず実行ファイル「sort」を削除しておきます．

```
[user@localhost sort]$ rm sort
[user@localhost sort]$ ls
Makefile  sort.c
[user@localhost sort]$
```

次に，「make」というコマンドを実行してみましょう．

```
[user@localhost sort]$ make
gcc sort.c -o sort
[user@localhost sort]$ ls
Makefile  sort  sort.c
[user@localhost sort]$
```

コンパイルが実行され，「sort」という実行ファイルが生成されました．

Makefileが正しく記述できていないと，もしかしたらmakeコマンドがエラーとなってしまうかもしれません．そのようなときは，以下の点を確認してみてください．

- **「gcc」や「rm」の前が，Tab文字でなく空白文字になっていないか？**
  この場合，以下のようなエラーになるようです．

```
Makefile:2: *** missing separator (did you mean TAB instead of 8 spaces?).  Stop.
```

- **Tabの前に空白文字などのようにして，空白文字が紛れ込んでいないか？**
  この場合，以下のようなエラーになるようです．

```
Makefile:2: *** missing separator.  Stop.
```

makeコマンドは，実行するとそのディレクトリにある「Makefile」というファイルを探し，Makefileに記述された処理を行います．

Makefileには**リスト14.1**にあるようにgccのコマンド実行手順が書かれており，gccが実行されてコンパイルが行われます．

14.1.3 Makefileをgitに登録しよう

リスト14.1のMakefileは，本章で改造して行きます．

ということでここで，Makefileもgitに登録しておきましょう．

```
[user@localhost sort]$ git add Makefile
[user@localhost sort]$ git commit Makefile
```

このようにMakefileもgitの管理対象として，登録することができます．

14.1.4 Makefileの書きかたは?

リスト14.1のMakefileについて，内容を説明しましょう．

まず先頭には「all :」と書かれた行があります．さらに「clean :」と書かれた行もあります．

これらはMakefileの「ターゲット」と呼ばれます．

例えば「make all」を実行すると，「all」というターゲットに記述された処理が行われます．

同様に「make clean」を実行すると，「clean」というターゲットの処理が実行されます．

ターゲットを指定せずに「make」を実行すると先頭のターゲット，つまり「all」を指定したものとされます．

試してみましょう.

```
[user@localhost sort]$ make clean
rm -f sort
[user@localhost sort]$ make all
gcc sort.c -o sort
[user@localhost sort]$ make
gcc sort.c -o sort
[user@localhost sort]$
```

「make clean」を実行することで rm コマンドにより実行ファイル「sort」が削除されます．これはいわばディレクトリの「掃除」に当たります.

そして「make all」を実行すると，再度 gcc が実行されます.

ターゲットを指定せずに「make」を実行した際には，「make all」と同様にコンパイルが行われています.

14.2 Makefileを整備しよう

ここまでではMakefileを作成することにあまりメリットはあるように感じられません.

そこで，Makefileを**リスト14.2**のように修正してみましょう.

リスト14.2: Makefileを修正する

```
001: all :   sort
002:
003: sort :  sort.c
004:         gcc sort.c -o sort
005:
006: clean :
007:         rm -f sort
```

前回と同様，行頭の空白の集まりはTabキーでタブ文字にすることに注意してください.

make cleanで掃除してから，makeを実行してみましょう.

```
[user@localhost sort]$ make clean
rm -f sort
[user@localhost sort]$ make
gcc sort.c -o sort
[user@localhost sort]$
```

同様にコンパイルができました.

14.2.1 ターゲットの依存関係とは何か?

リスト14.2では先頭にあるターゲットは「all」です．つまり「make」を実行すると，allターゲットの処理が行われることになります．しかし「all」の行の次は空行となっており，とくに処理は書かれていません.

かわりにallターゲットには，「all : sort」のようにして「sort」と記述されています.

これは「allターゲットの実施のためには，その前にsortターゲットの処理を行う必要がある」と

いうことを示します．

sortターゲットはその直後にあり，「gcc」のコマンド実行によりコンパイル処理を行うように記述されています．

このため単に「make」としてallターゲットを実施しようとすると，実際にはsortターゲットの処理が行われるわけです．

14.2.2 無駄な処理は行わない

makeコマンドの利点は，生成されたファイルの日付情報などを見て，処理の必要性を自動的に判断してくれることです．

例えばmake cleanで掃除した後，makeを連続して実行してみましょう．

```
[user@localhost sort]$ make clean
rm -f sort
[user@localhost sort]$ make
gcc sort.c -o sort
[user@localhost sort]$ make
make: Nothing to be done for `all'.
[user@localhost sort]$
```

２回目のmakeの実行時には「Nothing to be done」と出力されています．つまり，「何もやることが無い」という意味で，実際に何も行われていないようです．これはどういうことでしょうか．

この秘密は，sortターゲットの記述にあります．

sortターゲットは「sort : sort.c」のように書かれています．これは「sortというファイルの生成にはsort.cというファイルが必要である」という依存関係を示しています．

このような依存関係がある場合，makeコマンドは以下のようにして処理の実行の必要性を判断します．

- **sortがそもそも存在しない場合には，sortの生成処理を行う必要がある**
- **sortの日付とsort.cの日付を比較して，sortのほうが新しい場合には，sortは最新の状態なので処理を行う必要は無い**
- **sort.cのほうが新しい場合には，sort.cが更新されているので，再度処理を行ってsort を再生成する必要がある**

つまり1回目のmakeでsortが生成されましたが，2回目のmakeではsortがすでに生成済みであるため，コンパイル処理は不要と判断されたわけです．

このため例えばsort.cをエディタで修正し，再度makeを行ったような場合には，sortの再生成が必要と判断されることになります．やってみましょう．

```
[user@localhost sort]$ make
make: Nothing to be done for `all'.  (sortが生成済みのため，何も行われない)
[user@localhost sort]$ nano sort.c   (エディタでsort.cに変更を加える)
[user@localhost sort]$ make
gcc sort.c -o sort            (sort.cが更新されているため，sortを再生成する)
[user@localhost sort]$
```

もしくはtouchコマンドでsort.cの日付を更新しても，make時にsortが再生成されることになります．

```
[user@localhost sort]$ make
make: Nothing to be done for `all'.  (sortが生成済みのため，何も行われない)
[user@localhost sort]$ touch sort.c  (touchコマンドでsort.cの日付を更新する)
[user@localhost sort]$ make
gcc sort.c -o sort  (日付からsort.cが更新されていると判断し，sortを再生成する)
[user@localhost sort]$
```

このようにMakefileさえ適切に記述しておけば，処理の必要性をmakeコマンドが自動的に判断してくれます．

これは今回のようなシンプルな例ではあまり利点は感じられません．

しかし全コンパイルにかかる時間が数十分を超えるような巨大なプログラムになると，一部分のみ修正して実行ファイルを再生成するときにかかる時間を大幅に短縮できます．

修正版のMakefileを，gitに登録しておきましょう．

```
[user@localhost sort]$ git add Makefile
[user@localhost sort]$ git commit Makefile
```

14.3 Makefileで変数を利用してみよう

Makefileはここまでのような単純な例だけではなく，もっと複雑な記述をすることもできます．

例えば「変数」を使ってみましょう．「変数」はC言語でも利用しましたが，Makefile中でも似たようなものを使うことができます．

14.3.1 変数によって値を表現しよう

リスト14.2は，変数を利用して**リスト14.3**のように修正することができます．

リスト14.3: 変数を利用する

```
001: TARGET = sort
002:
003: all :   $(TARGET)
004:
005: $(TARGET) :     $(TARGET).c
006:         gcc $(TARGET).c -o $(TARGET)
007:
008: clean :
009:         rm -f $(TARGET)
```

リスト14.3は**リスト14.2**に対して，「sort」→「$(TARGET)」のような変更を加えたものです．これは，「TARGET」という変数を利用しています．

リスト14.3の先頭にある「TARGET = sort」は，「TARGET」という変数に「sort」という文字列を代入する，という意味です．その後の「$(TARGET)」という部分は変数の利用なので，代入された「sort」という文字列に置き換えられます．

よってその記述は，**リスト14.2**と等価になるわけです．

変数を使うことで，複数回利用される文字列を一箇所で定義し，文字列の変更時にはその定義部分のみ修正すればいいようにできます．例えば**リスト14.3**の例では，コンパイル対象のファイルを変更したいときには，TARGETの定義部分のみを変更すればいいことになります．

また変数を使うことには，その文字列がどのような意味を持つのかを指し示す，という大きな意味があります．例えば**リスト14.3**ならば，「TARGET」のように変数を定義することで，「$(TARGET)」としてある部分は「コンパイル対象のターゲットである」ということを指し示すことができます．

最後に完成版のMakefileを，gitに登録しておきましょう．

```
[user@localhost sort]$ git add Makefile
[user@localhost sort]$ git commit Makefile
```

14.4 フリーソフトウェアをコンパイルしてみよう

makeの仕組みはフリーソフトウェアのコンパイルなどにも応用されています．

試しになにか適当なフリーソフトウェアをダウンロードして，コンパイルしてみましょう．

なおここで説明する内容は，ＰＣがネットワークに接続されているという前提の話になります．もしもネットワーク接続されていない場合にはフリーソフトウェアのダウンロードが実施できないので，内容だけ読んで理解していただければと思います．

また実際にフリーソフトウェアなどをダウンロードして利用する場合には，その配布元が信用のおけるサイトなのか，十分に留意した上で行うようにしましょう．

14.4.1 フリーソフトウェアのソースコードをダウンロードしよう

ここではシンプルな「バイナリエディタ」として，「hexedit」というツールをコンパイルしてみます．

ひとまずホームディレクトリに戻りましょう．

```
[user@localhost sample]$ cd ../..
[user@localhost ~]$
```

さらにhexeditのソースコードとパッチをダウンロードします．

ダウンロードは「wget」というコマンドで行えます．以下の例では，筆者のホームページからダウンロードしています．

```
[user@localhost ~]$ wget http://kozos.jp/books/asm/hexedit-1.2.12.src.tgz
--2014-07-19 14:13:07--  http://kozos.jp/books/asm/hexedit-1.2.12.src.tgz
... (中略) ...
100%[======================================>] 65,802      --.-K/s   in 0.1s

2014-07-19 14:13:07 (666 KB/s) - `hexedit-1.2.12.src.tgz' saved [65802/65802]

[user@localhost ~]$
```

Column

インターネットにプロキシ経由で接続されている場合には，wgetの実行前に以下のようにしてプロキシの設定を行います．

（スーパーユーザでテキストエディタを起動し，wgetの設定ファイルを編集する）

```
[root@localhost user]# nano /etc/wgetrc
```

テキストエディタで設定ファイルを開いたら，以下のような行を追記します．（追記場所はどこでも構いませんが，とくに無ければファイル末尾でいいでしょう）

```
http_proxy = http://(プロキシのURL):(ポート番号)/
proxy_user =(プロキシのユーザ名:必要な場合のみ)
proxy_password =(プロキシのパスワード:必要な場合のみ)
```

hexedit向けのパッチもありますので，これもダウンロードしましょう．

```
[user@localhost ~]$ wget http://kozos.jp/patch/hexedit/hexedit-1.2.12-patch-2.zip
--2014-07-19 14:13:16--  http://kozos.jp/patch/hexedit/hexedit-1.2.12-patch-2.zip
... (中略) ...
100%[======================================>] 4,433       --.-K/s   in 0.04s

2014-07-19 14:13:16 (107 KB/s) - `hexedit-1.2.12-patch-2.zip' saved [4433/4433]

[user@localhost ~]$
```

なおhexeditと今回利用するパッチについては，以下を参照してください．

```
http://kozos.jp/patch/hexedit/index.html
```

ダウンロードが完了したら，これらを展開します．ひとまず以下のようにコマンド実行してみてください．

```
[user@localhost ~]$ tar xvzf hexedit-1.2.12.src.tgz
hexedit/hexedit.h
hexedit/hexedit.c
... (中略) ...
hexedit/configure.in
hexedit/Makefile-build.in
[user@localhost ~]$
```

hexeditのソースコードが展開できたので，どのようなファイルがあるのかを見てみましょう．

```
[user@localhost ~]$ cd hexedit
[user@localhost hexedit]$ ls
COPYING            TODO          display.c           hexedit.c   mark.c
Changes            config.h.in   file.c              hexedit.h   misc.c
Makefile-build.in  configure     hexedit-1.2.12.lsm  install-sh  page.c
Makefile.in        configure.in  hexedit.1           interact.c  search.c
[user@localhost hexedit]$
```

様々なファイルがあるようです．これらがhexeditのソースコードです．

しかし見たところ，Makefileは無いようです．これは後でプログラムで自動生成しますので，気にせずに進んでしまってください．

14.4.2 パッチを当てよう

さらに，パッチも展開します．現在はhexeditディレクトリの中にいますが，とりあえずカレントディレクトリに展開してしまいましょう．以下のようにコマンド実行してください．

```
[user@localhost hexedit]$ unzip ../hexedit-1.2.12-patch-2.zip
Archive:  ../hexedit-1.2.12-patch-2.zip
   creating: hexedit-patch/
  inflating: hexedit-patch/patch-display.c
  inflating: hexedit-patch/patch-hexedit.h
  inflating: hexedit-patch/patch-interact.c
  inflating: hexedit-patch/patch-search.c
  inflating: hexedit-patch/patch-hexedit.c
[user@localhost hexedit]$
```

パッチが展開できたら，それらをパッチ当てしてみましょう．これは8章で説明したpatchコマンドによって行えます．パッチは5つあるので，順番に当てていきます．

```
[user@localhost hexedit]$ patch < hexedit-patch/patch-display.c
patching file display.c
[user@localhost hexedit]$ patch < hexedit-patch/patch-hexedit.h
patching file hexedit.h
[user@localhost hexedit]$ patch < hexedit-patch/patch-interact.c
patching file interact.c
[user@localhost hexedit]$ patch < hexedit-patch/patch-search.c
patching file search.c
[user@localhost hexedit]$ patch < hexedit-patch/patch-hexedit.c
patching file hexedit.c
[user@localhost hexedit]$
```

これでソースコードの準備は完了です．

14.4.3 configureでMakefileを生成しよう

先述したように，hexedit は展開したままの状態では Makefile がありません．このためこのまま make を実行しても，エラーになってしまいます．

```
[user@localhost hexedit]$ ls Makefile*
Makefile-build.in  Makefile.in
[user@localhost hexedit]$ make
make: *** No targets specified and no makefile found.  Stop.
[user@localhost hexedit]$
```

Makefileは添付されているconfigureというプログラムを実行することで生成されますので，まずはconfigure を実行します．これは環境チェック用のプログラムで，環境に合わせてMakefile を自動生成してくれます．

このように「configure」が添付されていて，makeの前にconfigure を実行するというフリーソフトウェアは多く，コンパイルの定番の手順でもありますので，覚えておくといいでしょう．

```
[user@localhost hexedit]$ ./configure
checking for gcc... gcc
checking for C compiler default output file name... a.out
... (中略) ...
config.status: creating config.h
[user@localhost hexedit]$
```

configure を実行することで，Makefile が作成されます．見てみましょう．

```
[user@localhost hexedit]$ ls Makefile*
Makefile  Makefile-build  Makefile-build.in  Makefile.in
[user@localhost hexedit]$
```

14.4.4 makeでコンパイルしよう

Makefile が生成されたら，あとは make を実行するだけです．

```
[user@localhost hexedit]$ make
gcc -DHAVE_CONFIG_H -g -O2 -c hexedit.c
gcc -DHAVE_CONFIG_H -g -O2 -c display.c
... (中略) ...
gcc  -o hexedit hexedit.o display.o mark.o page.o file.o interact.o misc.o search.o
-lcurses
[user@localhost hexedit]$
```

複数に分割されたソースコードが順番にコンパイルされ，実行ファイルとして「hexedit」というファイルが生成されます．

実行ファイルが生成されたことを確認してみましょう．

```
[user@localhost hexedit]$ ls -l hexedit
-rwxrwxr-x. 1 user user 91760 Jul 19 14:22 hexedit
[user@localhost hexedit]$
```

14.4.5 実行しよう

生成された実行ファイルを起動してみましょう．hexeditはバイナリエディタで，指定したファイルをバイナリデータとして開いて編集することができます．

ためしに Makefile を hexedit で開いてみましょう．以下のようにコマンドライン引数に Makefile を指定して，hexedit を起動します．

```
[user@localhost hexedit]$ ./hexedit Makefile
```

図14.1のようにMakefileをバイナリデータとして見ることができれば成功です.

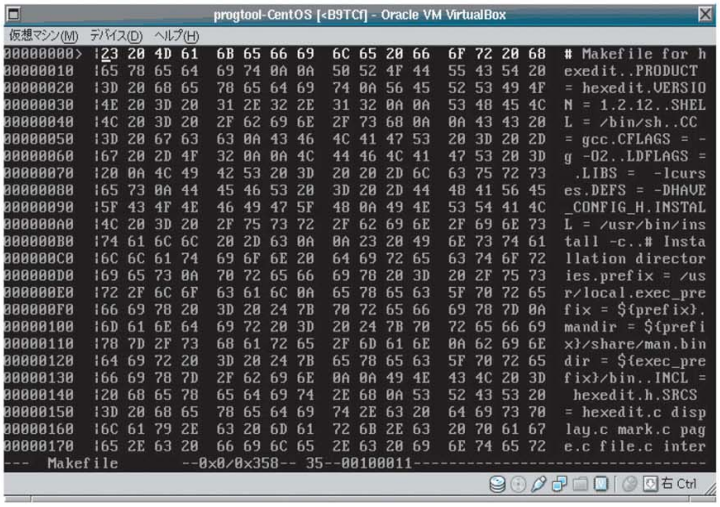


図14.1: hexeditでMakefileを開く

左側に出ているのはファイルの先頭からのオフセット値，中央はデータの16進数表記，右端はASCII文字による表示です．よくわからなかったら，まあこれはあくまで体験ですので，まあそんなもんか，くらいの理解でかまいません．

hexeditはCtrl＋Xで終了することができます.

ここではhexeditの使いかたを説明することは本題ではないので，hexeditについてこれ以上はとくに説明はしません．興味のあるかたはいろいろ操作してみるといいでしょう.

14.5 まとめ

makeは今回のようなプログラムのコンパイル処理に限らず，様々な処理の自動化に応用できます．実は本書の原稿も専用のMakefileを作成し，makeコマンドにより整形しています．うまく活用して様々な処理を自動化できると，その便利さを実感することができるでしょう．

またMakefileは「コンパイル手順が記述されているドキュメント」としての意味合いを持ちます．

例えばプログラムのソースコードを手渡す際に，そのためのMakefileも一緒に渡せば，それはコンパイル方法などの説明文書の代わりになりますから，そうした説明文書を新たに作成して手渡す必要は無くなるわけです．

●本書で紹介するCentOSのCUIコマンド－4（6～10章）

コマンド		ツールの役割	ページ
screen		複数のスクリーンを開くコマンド	148
	「Ctrl」+「J」[本書の場合]	エスケープキー　通常は「Ctrl」+「A」	150
	C	新しいスクリーンを開く	150
	P	ひとつ前のスクリーンに戻る	151
	N	ひとつ先のスクリーンに進む	151
	[	コピーモードに入る スペースキーで選択後、再度スペースキーでコピー	153
	]	ペースト	156
date		時刻を表示する	151
diff -u（ファイル名1）（ファイル名2）		（ファイル名1）（ファイル名2）の違いを出力する	170
diff -u（ファイル名1）（ファイル名2）>（パッチファイル）.patch		（ファイル名1）（ファイル名2）の違いを（パッチファイル）.patchに出力する	173
patch <（パッチファイル）.patch		（パッチファイル）.patchの内容を反映する ※（ファイル名2）の内容を（ファイル名1）に反映する	174
patch -R <（パッチファイル）.patch		（パッチファイル）.patchの内容を元に戻す ※（ファイル名1）の内容に戻す	175
git config --global user.name "（ユーザー名）"		git利用のユーザ名を登録	188
git config --global user.email メールアドレス		git利用のメールアドレスを登録	188
git config --global core.editor nano		git利用で使用するエディタとしてnanoを指定	189
git config --list		git設定内容を出力	189
git init		現在いるディレクトリをファイル管理のための「リポジトリ」に指定する	189
ls -a		「.」（ピリオド）が付加された隠しファイル（ディレクトリ）も含めたファイルの一覧を表示する	190

15

関数を使ってみよう

[C言語プログラミング編6]—関数

ここまででC言語のプログラムは，まず「main」と書き，その後の中括弧「{」と「}」の間にすべて書き込んできました．そしてこの中括弧の中に書かれた処理が，上から順番に実行されていました．

if文で条件分岐したり，for文やwhile文でループしたりもしましたが，「main() の中括弧の中にすべて書く」という点は同じでした．

しかしこのような書き方では，処理の内容が増えていったときにわかりにくくなってしまいます．すべて main() の中括弧の中に書くのでは，場合によっては何千行にも何万行にもなってしまうかもしれません．

そこで普通は，処理を分割して書くことになります．このようなものは「サブルーチン」と呼ばれます．特定の処理を切り出してサブルーチンにして，本処理からはサブルーチンを呼び出すようにするわけです．

サブルーチンは複数の箇所から呼び出すこともできます．このため同じ処理を複数箇所で行うような場合には，その処理はサブルーチン化しておけば，サブルーチンを呼び出すだけでその処理が行えるようになります．これはソースコード量を削減する効果があります．

C言語にはサブルーチンを実現する文法として「関数」があります．ここでは「処理を関数に分割する」ということを体験してみましょう．

15.1 ソート処理を関数化してみよう

13章のソート処理の完成形である，sort.c（**リスト13.5**）をもう一度見てみましょう．

リスト15.1: sort.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(int argc, char *argv[])
005: {
006:         int a[5] = { 5, 3, 4, 2, 1 };
007:         int x;
008:         int changed;
```

```
009:         int i;
010:
011:         changed = 1;
012:         while (changed == 1) {
013:                 printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
014:                 changed = 0;
015:                 for (i = 0; i < 4; i++) {
016:                         if (a[i] > a[i + 1]) {
017:                                 x = a[i + 1];
018:                                 a[i + 1] = a[i];
019:                                 a[i] = x;
020:                                 changed = 1;
021:                         }
022:                 }
023:         }
024:         exit(0);
025: }
```

これは14章でMakefileも作成しています． sort.c と Makefile は git で登録済みです．

```
[user@localhost ~]$ cd git/sort
[user@localhost sort]$ make clean
rm -f sort
[user@localhost sort]$ ls
Makefile  sort.c
```

復習として，もう一度 make によるコンパイルと実行を試しておきましょう．

```
[user@localhost sort]$ make
gcc sort.c -o sort
[user@localhost sort]$ ls
Makefile  sort  sort.c
[user@localhost sort]$ ./sort
5 3 4 2 1
3 4 2 1 5
3 2 1 4 5
2 1 3 4 5
1 2 3 4 5
[user@localhost sort]$
```

問題なく実行できています．

15.1.1 変数の交換処理を関数にしてみよう

リスト15.1でわかりにくいのは，変数の交換の部分です．これは以下のようになっています．

```
        x = a[i + 1];
        a[i + 1] = a[i];
        a[i] = x;
```

また sort.c は13章でいろいろ改造していますが，もともとはループを使わずに各変数ひとつひとつに対して比較処理をしていました．このため例えば**リスト13.4**では，上のような変数の交換処理は複数箇所に埋め込まれていました．

まずはこの部分を関数にして，処理を切り出してみましょう．

これは関数にすると，以下のように書くことができます．

```
void exchange(int a[], int i)
{
        int x;
        x = a[i + 1];
        a[i + 1] = a[i];
        a[i] = x;
}
```

なおここでは関数の作り方や文法については，あまり詳しくは説明しません．あくまで「体験してみること」を主眼に，チュートリアル的に説明します．

このため文法的なことがわからず，「なぜそう書けるのか？」と疑問に思うこともあるかもしれません．

しかしここではあまり気にせずに，あくまで雰囲気をつかむことだけに注力して，ひとまず「関数化」という流れを体験してみてください．

上の例では，「exchange()」という「関数」を定義しています．そして**リスト15.1**の sort.c を関数 exchange() を利用して書き換えると，**リスト15.2**のようになります．

リスト15.2: sort.c（関数版）

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: void exchange(int a[], int i)
005: {
006:         int x;
007:         x = a[i + 1];
008:         a[i + 1] = a[i];
009:         a[i] = x;
010: }
011:
012: int main(int argc, char *argv[])
013: {
014:         int a[5] = { 5, 3, 4, 2, 1 };
015:         int changed;
016:         int i;
017:
018:         changed = 1;
019:         while (changed == 1) {
020:                 printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
021:                 changed = 0;
022:                 for (i = 0; i < 4; i++) {
023:                         if (a[i] > a[i + 1]) {
024:                                 exchange(a, i);
025:                                 changed = 1;
026:                         }
027:                 }
028:         }
029:         exit(0);
030: }
```

このままでは変更点がわかりにくいですので，git diff の出力を見てみましょう．

```
@@ -1,10 +1,17 @@
 #include <stdio.h>
 #include <stdlib.h>

+void exchange(int a[], int i)
+{
+       int x;
+       x = a[i + 1];
+       a[i + 1] = a[i];
+       a[i] = x;
+}
+
```

```
 int main(int argc, char *argv[])
 {
        int a[5] = { 5, 3, 4, 2, 1 };
-       int x;
        int changed;
        int i;

@@ -14,9 +21,7 @@ int main(int argc, char *argv[])
                changed = 0;
                for (i = 0; i < 4; i++) {
                        if (a[i] > a[i + 1]) {
-                               x = a[i + 1];
-                               a[i + 1] = a[i];
-                               a[i] = x;
+                               exchange(a, i);
                                changed = 1;
                        }
                }
```

比較処理のブロック内にあった変数の値の交換処理が，exchange() という関数として切り出されていることがなんとなくわかるでしょうか．

15.1.2 関数には引数を渡すことができる

リスト15.2のプログラムには，まず exchange() という「関数」の定義を追加しています．さらに a[i] と a[i+1] の交換を行っていた部分は，関数 exchange() の呼び出しに置き換えています．

そして交換の中継に利用していた変数 x は不要になりましたので，定義を削除しています．

なお変数 i は，main() の内部と exchange() の内部の両方で利用されています．しかし変数の定義は関数ごとに行われているため，これらは異なる変数として扱われます．つまり main() の内部で利用されている「i」と，exchange() の内部で利用されている「i」は，異なる別物ということになります．

プログラムの実行が main() から行われることは，今までと同じです．プログラム中には main() よりも exchange() のほうが先に書かれていますが，それには関係なく，処理は main() の先頭から行われます．

そして処理が以下の部分にたどり着くと，関数 exchange() に記述してある処理が行われることになります．

```
        exchange(a, i);
```

つまり実行が一時的に exchange() のほうに飛ぶ感じになります．

関数には**リスト 15.2** のように，「引数」を渡すことができます．例えば**リスト 15.2** では，配列 a[] と変数 i の値を引数として関数 exchange() に渡しています．

exchange() では受け取った配列 a[] に対して，変数 i で示された位置の値を交換しています．

15.1.3 実行を確認しよう

値の交換を関数化したところで，実行を確認しておきましょう．

```
[user@localhost sort]$ make clean
rm -f sort
[user@localhost sort]$ make
gcc sort.c -o sort
[user@localhost sort]$ ./sort
5 3 4 2 1
3 4 2 1 5
3 2 1 4 5
2 1 3 4 5
1 2 3 4 5
[user@localhost sort]$
```

うまく動いたので，git で登録しておきましょう．

```
[user@localhost sort]$ git add sort.c
[user@localhost sort]$ git commit sort.c
```

15.2 GDBで関数の呼び出しを追ってみよう

関数呼び出しは，呼び出しによって実行が一時的に関数側に移り，関数内の処理が終了すると呼び出し元に戻ってくる，というイメージです．

これはちょっとわかりにくいかもしれませんが，こうした処理の流れは12章で説明した「GDB」で追うことができます．ここで復習がてら，関数呼び出しの動作をGDBで追ってみましょう．

15.2.1 デバッグオプションを付けてコンパイルしなおそう

GDBを利用するために，まずデバッグオプションである-gをつけてコンパイルしなおします．これにはMakefileを**リスト15.3**のように修正するといいでしょう．

リスト15.3: Makefileでデバッグオプションを指定する

```
001: TARGET = sort
002: CFLAGS = -g
003:
004: all :   $(TARGET)
005:
006: $(TARGET) :     $(TARGET).c
007:         gcc $(CFLAGS) $(TARGET).c -o $(TARGET)
008:
009: clean :
010:         rm -f $(TARGET)
```

git diffによる差分は以下のようになります．

```
@@ -1,9 +1,10 @@
 TARGET = sort
+CFLAGS = -g

 all :  $(TARGET)

 $(TARGET) :    $(TARGET).c
-       gcc $(TARGET).c -o $(TARGET)
+       gcc $(CFLAGS) $(TARGET).c -o $(TARGET)

 clean :
        rm -f $(TARGET)
```

CFLAGS という変数で，-g オプションを指定するように修正しました．これは例えばCFLAGS定義の先頭に以下のように「#」を付加すれば，無効にすることができます．Makefileでは「#」のついた行はコメントとして，無効として扱われるためです．

```
#CFLAGS = -g
```

このようにして，デバッグオプションの有効／無効を切り替えることができます．

Makefile を修正したところで，git に登録しなおしておきましょう．

```
[user@localhost sort]$ git add Makefile
[user@localhost sort]$ git commit Makefile
```

さらに make でコンパイルしなおします．

```
[user@localhost sort]$ make clean
rm -f sort
[user@localhost sort]$ make
gcc -g sort.c -o sort
[user@localhost sort]$
```

今度は -g オプションが付加されてコンパイルされています．

15.2.2 gdbを起動しよう

では，gdbを起動しましょう．

```
[user@localhost sort]$ gdb -tui sort
```

起動して適当にEnterキーを押すと，**図15.1** のような画面になります．「(gdb)」のプロンプトが表示されて，コマンド待ちの状態になっています．

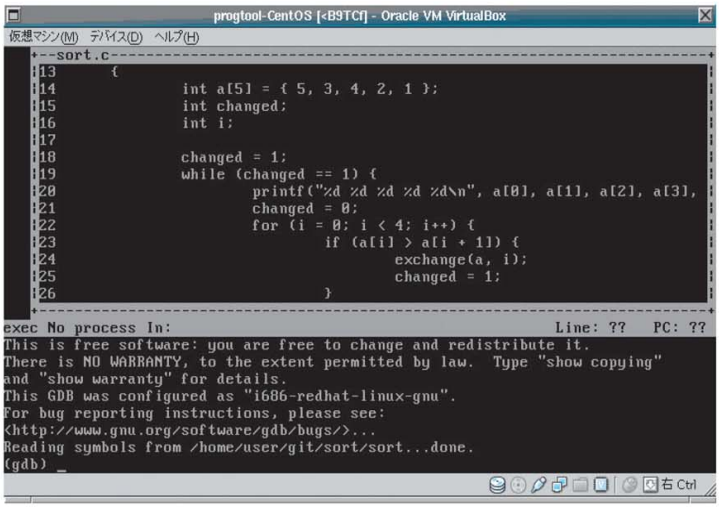


図 15.1: gdb を起動する

プロンプトから以下を実行して，main() の先頭にブレークポイントを設定して実行を開始しましょう．

```
(gdb) break main
(gdb) run
```

画面は**図 15.2** のようになり，main() の先頭がハイライトされています．main() の先頭にブレークポイントが設定されているため，先頭で処理が一時停止しています．

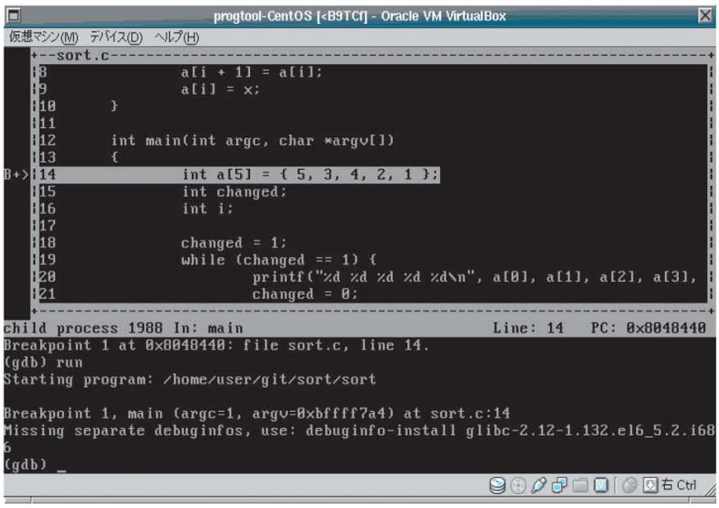


図 15.2: 実行を開始する

15.2.3 ステップ実行を進めよう

ここで「next」を実行し，ステップ実行をしていきます．

```
(gdb) next
```

ステップ実行を何度か繰り返すといずれ**図15.3**のように，exchange() の呼び出し部分にさしかかります．

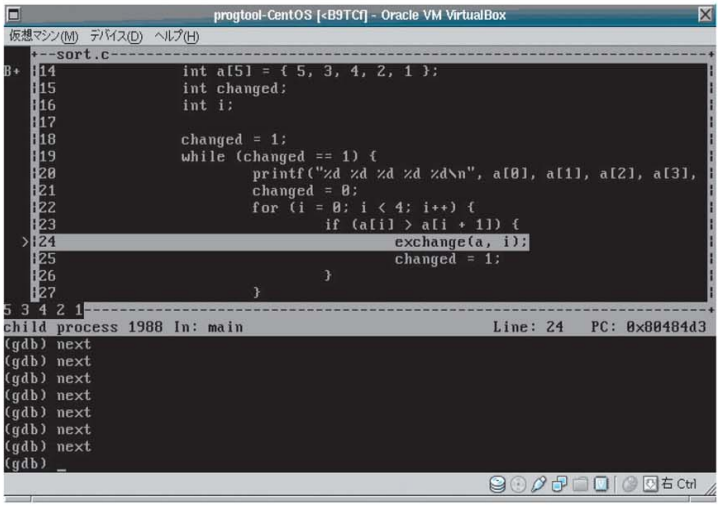


図15.3: exchange() を呼び出している箇所にさしかかる

ここで今まで通り「next」を実行してみます．

```
(gdb) next
```

すると**図15.4**のようにして，exchange() の呼び出しの次の行まで実行が進みます．

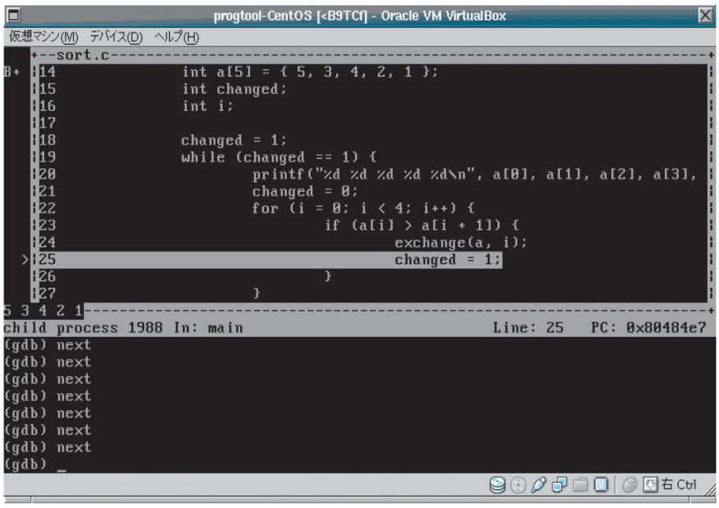


図 15.4: next では関数呼び出しの次の行まで一気に実行される

これは「next」によるステップ実行が，関数呼び出しも 1 行として扱うためです．このため関数の内部を実行し，関数を呼び出した箇所に戻るまでを一気に進めてしまっています．

15.2.4 関数の内部の処理を見てみよう

実はステップ実行には，「next」以外にも「step」というものがあります． next に対して step だと，関数の内部の処理に入っていきます．

次は step によるステップ実行を試してみましょう．その前に，もう一度 next を繰り返して処理を進めます．

```
(gdb) next
```

next を数回実行すると，**図 15.5** のように exchange() の呼び出しまで，再びたどり着きます．**図 15.3** と同じ状態です．

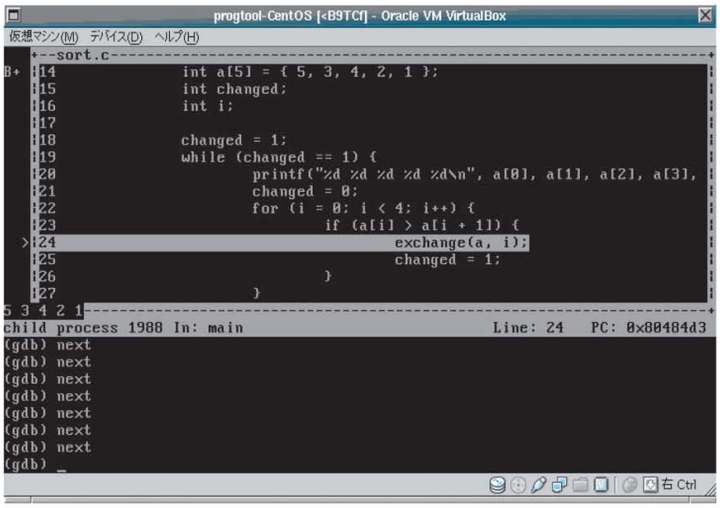


図15.5: exchange()の呼び出し箇所に再びたどりつく

ここで，今度は「step」を実行してみましょう．

```
(gdb) step
```

すると画面は**図15.6**のようになります． exchange()の内部がハイライトされています．
つまり処理がexchange()の内部に移っているわけです．

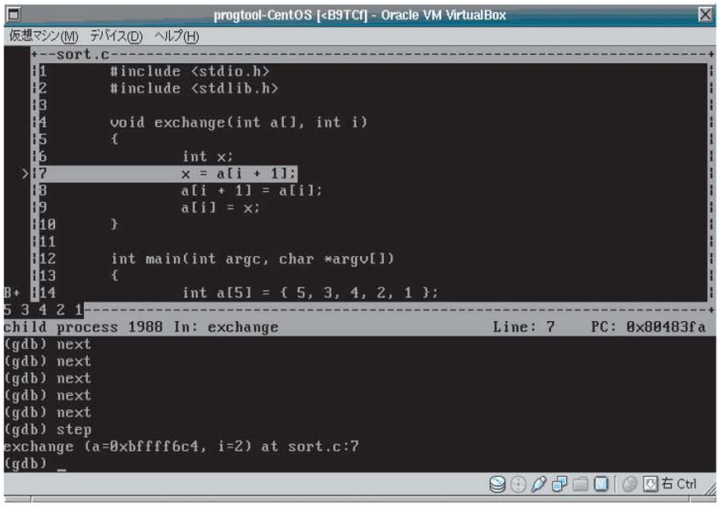


図15.6: stepで関数の中の処理に入っていく

exchange()の中でnextを実行してみましょう．

```
(gdb) next
```

図 15.7 のようにして，ハイライトされている箇所が進みます．

つまり関数exchange()の内部で処理が進んでいるわけです．

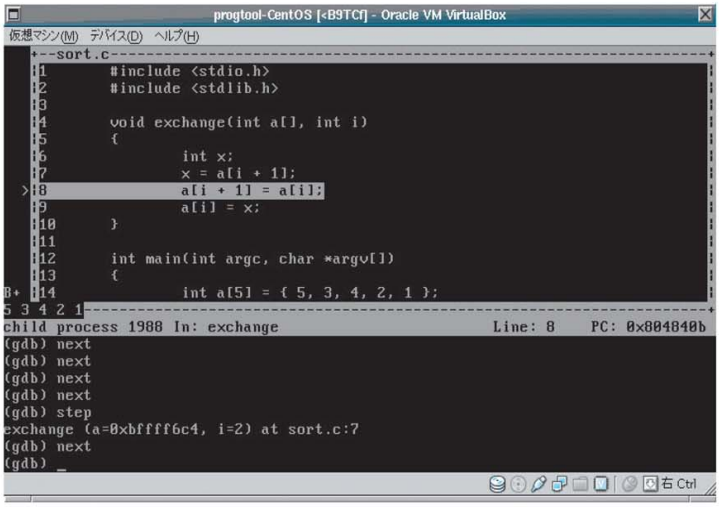


図 15.7: 関数内部でステップ実行を進める

さらにステップ実行を進めると，**図 15.8** のようにして exchange()の呼び出し箇所に処理が戻ります．

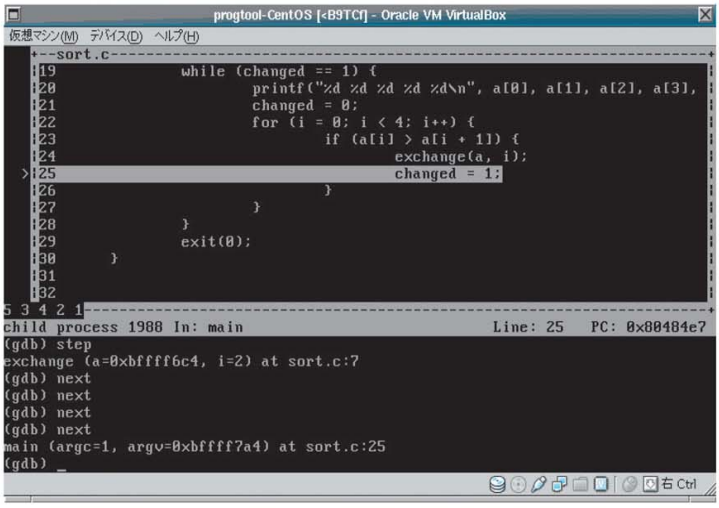


図 15.8: 関数の呼び出し元に戻る

つまり，関数呼び出しが完了して呼び出し元に戻ってきているわけです．

15.2.5 配列の内容を確認しよう

ここで print によって配列a[]の内容を確認してみましょう．

```
(gdb) print a
```

実行すると**図15.9**のようになります．

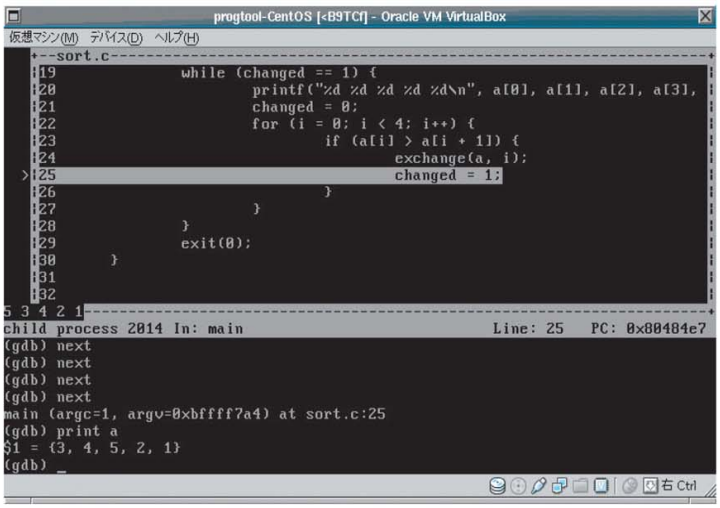


図15.9: 配列の内容を確認する

プログラム中では，配列a[]は以下のように初期化されています．

```
        int a[5] = { 5, 3, 4, 2, 1 };
```

これに対して**図15.9**では，配列a[]の内容は以下のようになっています．

```
$1 = {3, 4, 5, 2, 1}
```

「5」が後ろのほうに進み，そのぶんだけ「3」と「4」が前に出てきています．つまりソート処理が多少進んで，部分的に並び替えが行われているようです．

15.3 関数を多段に呼び出してみよう

関数は，関数の内部から呼び出すこともできます．

さらに関数化を進めて，関数の中から関数を呼び出すような構造を試してみましょう．

15.3.1 比較処理を関数にしてみよう

リスト15.2はwhileループの中にforループが入っています．このような構造は「多重ループ」と呼ばれます．

多重ループは見通しが悪くなるので，内側のループ処理も関数として切り出してみましょう．

すると，**リスト15.4**のようになります．

リスト15.4: sort.c（関数版その2）

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: void exchange(int a[], int i)
005: {
006:         int x;
007:         x = a[i + 1];
008:         a[i + 1] = a[i];
009:         a[i] = x;
010: }
011:
012: int compare(int a[], int n)
013: {
014:         int changed;
015:         int i;
016:         changed = 0;
017:         for (i = 0; i < n - 1; i++) {
018:                 if (a[i] > a[i + 1]) {
019:                         exchange(a, i);
020:                         changed = 1;
021:                 }
022:         }
```

```
023:         return changed;
024: }
025:
026: int main(int argc, char *argv[])
027: {
028:         int a[5] = { 5, 3, 4, 2, 1 };
029:         int changed;
030:
031:         changed = 1;
032:         while (changed == 1) {
033:                 printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
034:                 changed = compare(a, 5);
035:         }
036:         exit(0);
037: }
```

リスト15.4では比較処理の部分を切り出して，以下のように関数化することで「compare()」という関数に仕立てあげています．

```
int compare(int a[], int n)
{
        int changed;
        int i;
        changed = 0;
        for (i = 0; i < n - 1; i++) {
                if (a[i] > a[i + 1]) {
                        exchange(a, i);
                        changed = 1;
                }
        }
        return changed;
}
```

15.3.2 関数からは戻り値を返すことができる

比較処理は，「値の交換が行われたか？」という結果を返す必要があります．これは13章で説明した，処理の終了の検知のためです．

そして関数は「戻り値」を返すことができます．関数compare()の末尾には，以下のような行があります．

```
        return changed;
```

ここで変数changedの値を戻すことで，「値の交換が行われたか？」という結果を返しています．

リスト15.4では，関数compare()の呼び出し元は以下のようになっています．

```
        changed = compare(a, 5);
```

これでreturnによって返された値を，changedという変数で受け取ることになります．

これによりcompare()の内部で値の交換が行われたことを受け取り，ループを継続するか，終了するかを判断しているわけです．

実行してみましょう．

```
[user@localhost sort]$ make
gcc -g sort.c -o sort
[user@localhost sort]$ ./sort
5 3 4 2 1
3 4 2 1 5
3 2 1 4 5
2 1 3 4 5
1 2 3 4 5
[user@localhost sort]$
```

うまく実行できました．

最後に，完成版のsort.cをgitで登録しておきましょう．

```
[user@localhost sort]$ git add sort.c
[user@localhost sort]$ git commit sort.c
```

15.4 まとめ

本章の例では変数の値の交換処理を関数化しています．これがもしも複数の箇所で行われている処理ならば，関数化することで，プログラム量を減らすメリットがあります．

しかし本章の例では，一箇所で行われている処理を関数化しただけで，プログラム量を減らすメリットはありません．むしろ関数定義のためにプログラムの行数は増えていると言えるでしょう．

しかしそれでも，関数にすることにはメリットがあります．たとえば本章の例では，変数の交換部分を「exchange()」という名前で関数化することで，処理に「exchange」という「名前」がつけられています．

これには「処理に名前をつけて意味を持たせる」という効果があります．これはプログラムを他人にわかりやすく書く，という点で重要な意味を持ちます．

●本書で紹介するCentOSのCUIコマンドー5（10～14章）

コマンド	ツールの役割	ページ
git add（ファイル名）	（ファイル名）をgitに登録する	190
git commit（ファイル名）	（ファイル名）をコミットする ※登録したエディタが起動するのでコメントを入力して保存する	190
git log（ファイル名）	（ファイル名）の履歴を見る	193
git checkout（ファイル名）	（ファイル名）を元に戻す	194
git diff（ファイル名）	リポジトリに登録済みの（ファイル名）に対して、現在の（ファイル名）の差分を出力	195
git mv（ファイル名）（ディレクトリ名）	gitに登録した（ファイル名）を（ディレクトリ名）に移動する	201
gcc（Cプログラムファイル名） -o（実行ファイル名）-g	デバッグオプションをつけてコンパイルする	216
gdb -tui（実行ファイル名）	（実行ファイル名）のデバッグを行う	217
run	デバック中のプログラムを実行する	218
break（関数名）	（関数名）実行の直前にブレークポイントを設定する	220
next	（ブレークポイントで）一時停止状態からステップ実行する	222
print（変数名）	（変数名）の中身を表示する	223
step	関数の中の処理に入っていく	281
continue	（ブレークポイントで）一時停止状態を解除して実行再開	228
quit	gdbから抜ける	228
make	Makefileの記述内容（先頭のターゲット）を実行	252
make（ターゲット名）	Makefileの記述内容（ターゲットの内容）を実行	254
wget（ファイルのURL）	（ファイルのURL）にあるファイルを取得	261
tar xvzf（TARファイル）.tgz	（TARファイル）.tgzを展開	262
unzip（ZIPファイル）.zip	（ZIPファイル）.zipを展開	263, 324
./configure	ダウンロードしたファイルからMakefileを自動生成してくれる環境チェック用のプログラム	264

16

スクリプト言語を書いてみよう
[ツール編7]—Perl

ここまで，いくつかのプログラムをＣ言語で書いてみました．

しかしＣ言語で書く場合，実行の際には必ずコンパイル→実行という手順を踏む必要があります．Makefileを作成して自動化はしましたが，そもそも「コンパイル」が必要ということを「面倒」と感じた読者のかたもいるのではないでしょうか．

コンピュータ言語は様々なものがあるのですが，その中には「スクリプト言語」という便利なものがあります．

ここでは「Perl」というスクリプト言語に触れてみましょう．

16.1 Perlで「ハローワールド」を書いてみよう

Perlは「パール」と呼ばれる，スクリプト言語のひとつです．

その前にそもそも「スクリプト言語」とは何でしょうか？

16.1.1 スクリプト言語とは何か?

Ｃ言語は「コンパイル」という作業により，そのＰＣで実行するための機械語コードに変換されます．このような言語は「コンパイラ型」と呼ばれます．「ソースコード編集」→「コンパイル」→「実行」のようにして，実行までに「コンパイル」という手順を踏む必要があります．

これに対してスクリプト言語は，様々な命令列（スクリプト）を解釈して当該の処理を実行するプログラムを始めからインストールしておき，その解析プログラムがスクリプトを読んで実行するというものです．

このような処理はインタプリタ型とも呼ばれ，コンパイルの必要無くソースコードをいきなり実行できるという手軽さがあります．

反面，ソースコードを解析しながら実行するため速度的に劣るという話があるのですが，それも今は昔の話で，現在は解析処理の技術が向上したり，そもそもＰＣが十分に速いため，「スクリプト言語は遅い」というのは過去の話となっています．

むしろこれらのスクリプト言語はLL（Lightweight Language:軽量プログラミング言語）などと

呼ばれ，人気のある分野です．（ここで言う「軽量」は，コンピュータにとって負荷が低いという意味ではなく，人間にとって書きやすい（プログラミングの労力が少なく済む）という意味です）

筆者が本書での説明にPerlを選んだのは，多くのシステムで標準的にインストールされているスクリプト言語だからです．

スクリプト言語には他にもRuby（ルビー）やPython（パイソン）といったものがあります．しかしこれらはシステムに標準的にインストールされていないかもしれません．その点，Perlは多くのシステムで標準的に使えますので，覚えておいて損は無いでしょう．

16.1.2 Perlで Hello World を書いてみよう

5章で説明したC言語の入門では，「Hello World!」と表示するプログラムを書いてみました．定番の，いわゆる「ハローワールド」です．

Perlでまずは定番の「Hello World」を書いてみると，**リスト16.1**のようになります．

リスト16.1: hello.pl

```
001: #!/usr/bin/perl
002:
003: print "Hello World!\n";
```

リスト16.1をhello.plというファイル名で作成します．これが，「スクリプト」と呼ばれるものです． Perlのスクリプトは，拡張子は「.pl」にするのが通例です．

さらに以下を実行して，hello.plに実行属性を付加しておきます．これは，最初の1回だけ行っておけば大丈夫です．

```
[user@localhost ~]$ chmod +x hello.pl
```

実行属性が付加されたどうかを確認しましょう． lsを-lというオプションを付けて実行します．

```
[user@localhost ~]$ ls -l hello.pl
-rwxrwxr-x. 1 user user 41 Jul 20 16:57 hello.pl
[user@localhost ~]$
```

行頭の「-rwxrwxr-x」の部分を確認してください．ここで「x」という表記がついていたら大丈夫

です．

では，実行してみましょう．スクリプトは，それ自身を実行ファイルとして直接実行することができます．

```
[user@localhost ~]$ ./hello.pl
Hello World!
[user@localhost ~]$
```

「Hello World!」と表示されました．うまく実行できています．

もしも以下のようになった場合には，実行属性が付加されていないかもしれません．もう一度 chmod を実行して，再度確認してみてください．

```
[user@localhost ~]$ ./hello.pl
-bash: ./hello.pl: Permission denied
[user@localhost ~]$
```

16.1.3 C言語と比較してみよう

リスト 16.1 の Perl スクリプトは，5 章で説明した，**リスト 5.1** の C 言語のプログラム（hello.c）と実行結果は同じです．

つまり（実行の手順などは別として，結果だけを見るならば）等価なプログラムと言えるわけです．

しかし C 言語で書いた「hello.c」（**リスト 5.1**）は，全体で 8 行もあります．やりたいことは「Hello World!」と表示することだけなのに，これはちょっと多いようにも思えます．また「#include」や「main()」といったお決まりのおまじないのような記述もあり，そうした行がプログラムの行数を増やしています．

対して Perl で書いた「hello.pl」（**リスト 16.1**）は，たったの 3 行です．先頭には「#!/usr/bin/perl」というおまじないのような行もありますがそれもたったの 1 行で，あとは 3 行目に「Hello World と表示する」という 1 行があるだけです．つまり，プログラム中で本当にやりたいことが，プログラムの大部分を占めていることになります．

スクリプト言語の良いところはその手軽さにもありますが，「同じことを少ないプログラム量で実現できる」というところにもあります．

16.1.4 ではなぜC言語を使うのか?

ここまで読むと,「ではすべてPerlでいいではないか」「なぜC言語が必要なのか?」という疑問が湧くかもしれません.

そもそも以前はスクリプト言語は「実行速度が遅い」というイメージが強くありました.しかし現在は速度向上により,実用的に問題となることも少なくなりました.またスクリプト言語は開発のしやすさのメリットが大きくあります.単に実行速度だけで開発言語を選択することもナンセンスな時代になっています.

ただやはりそれでも速度が必要なところや,CPUのハードウェアにより近い操作を行う必要があるプログラム(OSなど)では,相変わらずC言語が必要です.

以前は「10年後にはプログラミング言語はすべて統一され,他の言語は淘汰される」というように集束方向に向かうように言われてもいました.

しかし実際にはプログラミング言語は多種多様に進化し,先述した「LL」といった言葉も生まれています.

これはPCの速度向上により実行速度だけで言語を選択する必要性が薄くなったこと,プログラミングが様々な分野に普及し各分野に適切な言語が選択されるようになったこと,ソフトウェアが高度化し,少量のソースコードで多彩な機能が実現できることの要望が増えたことなどにより,多種の言語が統合されるのではなくより細分化し,住み分けされたためのように思います.

これは時代が「コンピュータの性能不足」という点から,「いかにやりたいことを実現するか?」という目的を意識したものにシフトした結果のように思います.

つまりより人間向けに技術が進化した結果であって,喜ばしいことのように筆者は感じます.

16.2 Perlでソートのプログラムを書いてみよう

まずはPerlに慣れるためのお試しとして「Hello World」を書いてみたわけですが，もう少し複雑なプログラムを作成してみましょう．

13章では配列の利用の練習として「ソート」のプログラムをC言語で書いてみました．
リスト13.5の「sort.c」が13章でのソートの完成形ですが，ここでは**リスト13.5**のプログラムをPerlで書きなおしてみましょう．

なお，実はPerlにはソートを行うための機能が標準であります．またC言語でも「qsort()」というライブラリ関数を利用することで，ソートを行うことができます．これらを利用すれば楽にソートを実現できます．

しかし本書ではプログラミング体験と比較のために，あえてそのようなものは使わずにソートの処理を自身でプログラミングしています．

16.2.1 まずはそのままPerlに変換してみよう

まずは**リスト13.5**のプログラムで行っている処理を，C言語からPerlにそのまま書き換えてみます．ここに**リスト13.5**のプログラムを，**リスト16.2**としてもう一度掲載しておきます．

リスト16.2: hello.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: int main(int argc, char *argv[])
005: {
006:         int a[5] = { 5, 3, 4, 2, 1 };
007:         int x;
008:         int changed;
009:         int i;
010:
011:         changed = 1;
012:         while (changed == 1) {
013:                 printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
014:                 changed = 0;
015:                 for (i = 0; i < 4; i++) {
```

```
016:                        if (a[i] > a[i + 1]) {
017:                                x = a[i + 1];
018:                                a[i + 1] = a[i];
019:                                a[i] = x;
020:                                changed = 1;
021:                        }
022:                }
023:        }
024:        exit(0);
025: }
```

次に**リスト16.2**のC言語のプログラムを，Perlに変換してみます．すると**リスト16.3**のようになりました．

リスト16.3: sort.pl

```
001: #!/usr/bin/perl
002:
003: @a = ( 5, 3, 4, 2, 1 );
004:
005: $changed = 1;
006: while ($changed == 1) {
007:        print "$a[0] $a[1] $a[2] $a[3] $a[4]\n";
008:        $changed = 0;
009:        for ($i = 0; $i < 4; $i++) {
010:                if ($a[$i] > $a[$i + 1]) {
011:                        $x = $a[$i + 1];
012:                        $a[$i + 1] = $a[$i];
013:                        $a[$i] = $x;
014:                        $changed = 1;
015:                }
016:        }
017: }
```

説明は後にして，まずはソートのPerl版を実行してみましょう．**リスト16.3**をsort.plという名前で作成し，以下のようにして実行属性を付加します．

```
[user@localhost ~]$ chmod +x sort.pl
```

動作させてみましょう．

```
[user@localhost ~]$ ./sort.pl
5 3 4 2 1
3 4 2 1 5
3 2 1 4 5
2 1 3 4 5
1 2 3 4 5
[user@localhost ~]$
```

これは**リスト 13.5** を実行したときと同じ実行結果です．

つまりPerlでも，等価なプログラムを作成できているわけです．

16.2.2 PerlとC言語の違いは何か?

C言語によるソートのプログラムをPerlに変換する上で書きなおした点について，説明しましょう．

まず**リスト 16.2** の 1 ～ 2 行目にあった「#include」という行は，C言語特有の表記です．Perlでは不要なため，これは削除します．

さらにC言語では，プログラムの実行が開始する箇所として「main()」という記述をしていました．

しかしPerlではそのような記述は必要無く，書かれている処理が先頭から順に実行されます．やりたい処理をいきなり書き出してしまって構わないわけです．このためC言語のプログラムで書かれていた「int main ...」の行は削除しています．

また変数は「$i」のようにして，先頭に「$」を付けます．配列の初期化の表現も異なるため，Perlに合わせて書き換えました．

そのようにhello.cを全体的にPerlの書式に合わせて書き直したものが，**リスト 16.3** のsort.plです．ソートのアルゴリズムは同じため，内容的には「同じプログラムである」と言えます．

ソートのPerl版ができたところで，これも git で登録しておきましょう．

```
[user@localhost ~]$ mkdir git/sort-perl
[user@localhost ~]$ cd git/sort-perl
[user@localhost sort-perl]$ mv ../../sort.pl .
[user@localhost sort-perl]$ git add sort.pl
[user@localhost sort-perl]$ git commit sort.pl
```

16.3 Perlのプログラムを改良しよう

これだけ見ると，Perlでもそれほど短く書けるわけではないようにも思います．

しかしPerlでは，様々な記述の工夫ができます．ここでは可能な限りソースコードを短くしていくことに挑戦してみましょう．

16.3.1 可能な限り短くしてみよう

まずC言語では，変数の値の交換は以下のようにして，仲介用の変数を一時的に介して実現していました．

```
x = b;
b = a;
a = x;
```

しかしPerlでは，値の交換は以下のようにして1行で行えます．

```
($a, $b) = ($b, $a);
```

これを利用して**リスト16.3**のsort.plを書き換えてみると，**リスト16.4**のようになりました．

リスト16.4: 値の交換を1行にまとめる

```
001: #!/usr/bin/perl
002:
003: @a = ( 5, 3, 4, 2, 1 );
004:
005: $changed = 1;
006: while ($changed == 1) {
007:         print "$a[0] $a[1] $a[2] $a[3] $a[4]\n";
008:         $changed = 0;
009:         for ($i = 0; $i < 4; $i++) {
010:                 if ($a[$i] > $a[$i + 1]) {
011:                         ($a[$i], $a[$i + 1]) = ($a[$i + 1], $a[$i]);
012:                         $changed = 1;
013:                 }
014:         }
015: }
```

どうでしょうか？これだけでも，十分にすっきりしたように思えます．

git diff で変更点を確認してみましょう．

```
[user@localhost sort-perl]$ git diff sort.pl
diff --git a/sort-perl/sort.pl b/sort-perl/sort.pl
index 9354f60..ce18224 100755
--- a/sort-perl/sort.pl
+++ b/sort-perl/sort.pl
@@ -8,9 +8,7 @@ while ($changed == 1) {
        $changed = 0;
        for ($i = 0; $i < 4; $i++) {
                if ($a[$i] > $a[$i + 1]) {
-                       $x = $a[$i + 1];
-                       $a[$i + 1] = $a[$i];
-                       $a[$i] = $x;
+                       ($a[$i], $a[$i + 1]) = ($a[$i + 1], $a[$i]);
                        $changed = 1;
                }
        }
[user@localhost sort-perl]$
```

仲介用の変数「$x」を利用して３行で書いていた部分を，１行にまとめることができています．２行の削減になっています．

修正版として，これを新しく登録しておきましょう．

```
[user@localhost sort-perl]$ git add sort.pl
[user@localhost sort-perl]$ git commit sort.pl
```

16.3.2 終了のチェックを工夫してみよう

リスト 16.4 では変更があったかどうかの確認に，$changed という変数を利用しています．

これはＣ言語のプログラムでもそうなっていたものをそのまま流用しているわけなのですが，これをもっと簡単に検知する方法は無いでしょうか？

調べたいのは「配列に変更があった」ということです．

そして配列の全体を保存し，さらに比較することができるならば，次ページのようにして「配列に変更があったかどうか」を調べることができます．

- 配列を変更前に，別の配列にそのままコピーしておく
- 変更前の配列と変更後の配列を比較し，違いがあるかどうかを調べる

リスト16.4のsort.plをそのように書き換えてみると，**リスト16.5**のようになりました．

リスト16.5: 配列を比較する

```
001: #!/usr/bin/perl
002:
003: @a = ( 5, 3, 4, 2, 1 );
004:
005: while (join(',', @a) ne join(',', @b)) {
006:         print "$a[0] $a[1] $a[2] $a[3] $a[4]\n";
007:         @b = @a;
008:         for ($i = 0; $i < 4; $i++) {
009:                 if ($a[$i] > $a[$i + 1]) {
010:                         ($a[$i], $a[$i + 1]) = ($a[$i + 1], $a[$i]);
011:                 }
012:         }
013: }
```

リスト16.5では配列@bに配列@aの内容をコピーし，while文のループ条件ではそれらが一致するかどうかを調べています．なおjoin()は配列の内容を連結する処理です．連結後の文字列を比較することで，配列の一致性を確認しているわけです．ですが，あくまでこれはPerlの体験ですので，まあよくわからなければそれでも構いません．

git diffで変更点を確認してみましょう．

```
[user@localhost sort-perl]$ git diff sort.pl
diff --git a/sort-perl/sort.pl b/sort-perl/sort.pl
index ce18224..397da7d 100755
--- a/sort-perl/sort.pl
+++ b/sort-perl/sort.pl
@@ -2,14 +2,12 @@

 @a = ( 5, 3, 4, 2, 1 );

-$changed = 1;
-while ($changed == 1) {
+while (join(',', @a) ne join(',', @b)) {
         print "$a[0] $a[1] $a[2] $a[3] $a[4]\n";
-        $changed = 0;
+        @b = @a;
```

```
        for ($i = 0; $i < 4; $i++) {
                if ($a[$i] > $a[$i + 1]) {
                        ($a[$i], $a[$i + 1]) = ($a[$i + 1], $a[$i]);
-                       $changed = 1;
                }
        }
 }
[user@localhost sort-perl]$
```

2行の削減になっています.

そして**リスト 16.5**ではプログラム自体もすっきりしましたが,「$changed」というフラグを利用せずに「配列が等しくなければ」という条件でループできています.

つまり, より処理の内容を的確に表現しているプログラムになっているとも言えるでしょう.

これも, gitで登録しておきます.

```
[user@localhost sort-perl]$ git add sort.pl
[user@localhost sort-perl]$ git commit sort.pl
```

16.3.3 if文の後置表記を利用してみよう

これだけで処理はだいぶシンプルになりましたが, 実はPerlではif文は後置することで, シンプルに書くことができます.

例えば以下のようなif文があったとします.

```
if () {
  ...;
}
```

これはPerlでは, 実は以下のように書くことができます.

```
... if ();
```

このように書くことで「{…}」の中括弧が不要になり, さらにシンプルになります.
これを利用して書き換えると, sort.plは**リスト 16.6**のようになります.

リスト16.6: if文の後置表記を使う

```
001: #!/usr/bin/perl
002:
003: @a = ( 5, 3, 4, 2, 1 );
004:
005: while (join(',', @a) ne join(',', @b)) {
006:         print "$a[0] $a[1] $a[2] $a[3] $a[4]\n";
007:         @b = @a;
008:         for ($i = 0; $i < 4; $i++) {
009:                 ($a[$i], $a[$i + 1]) = ($a[$i + 1], $a[$i])
010:                         if ($a[$i] > $a[$i + 1]);
011:         }
012: }
```

これは処理の内容を工夫しているわけではなく単にプログラムの記述上の話です．しかしこのような後置の書き方はより簡潔な表現になるため，人気のある方法でもあります．

git diff で変更点を確認してみましょう．

```
[user@localhost sort-perl]$ git diff sort.pl
diff --git a/sort-perl/sort.pl b/sort-perl/sort.pl
index 397da7d..c881614 100755
--- a/sort-perl/sort.pl
+++ b/sort-perl/sort.pl
@@ -6,8 +6,7 @@ while (join(',', @a) ne join(',', @b)) {
        print "$a[0] $a[1] $a[2] $a[3] $a[4]\n";
        @b = @a;
        for ($i = 0; $i < 4; $i++) {
-               if ($a[$i] > $a[$i + 1]) {
-                       ($a[$i], $a[$i + 1]) = ($a[$i + 1], $a[$i]);
-               }
+               ($a[$i], $a[$i + 1]) = ($a[$i + 1], $a[$i])
+                       if ($a[$i] > $a[$i + 1]);
        }
 }
[user@localhost sort-perl]$
```

1行の削減になっています．

16.3.4 プログラムの動作を確認しよう

最後に，改造後も問題なく実行できることを確認しておきましょう．

```
[user@localhost ~]$ ./sort.pl
5 3 4 2 1
3 4 2 1 5
3 2 1 4 5
2 1 3 4 5
1 2 3 4 5
[user@localhost ~]$
```

うまく動作しています．

初期の**リスト 16.3**は17行ありました．これに対して最終版の**リスト 16.6**は，12行です．全体で5行の削減ができています．

これは「5行」という数字だけをみるとたいしたことはないようにも思えてしまいますが，ソースコードの割合として見ると，実に30%の削減です．そうして考えると，大きな効果とも考えられるのではないでしょうか．

最後にこれを，完成版としてgitで登録しておきましょう．

```
[user@localhost sort-perl]$ git add sort.pl
[user@localhost sort-perl]$ git commit sort.pl
```

16.4 まとめ

C言語で書いたプログラムとPerlで書いたプログラムを見比べてみると，C言語はいかにも不便で，Perlが便利であるかのようにも思えてしまいます．

しかしこれらは，プログラミング言語の「目的の違い」によります．

C言語は，コンピュータが理解する「機械語」をいかに移植性高く表現するか？という考えで設計されています．このためC言語では，処理の内容を機械語にそのまま変換できることが重要視されています．

たとえば複数の値を同時に扱うような命令は，コンピュータが標準的に持っているものではありません（そのような機能を持っているCPUもありますが，標準的とは言えません）．このためC言語では，複数の値を一括して扱うような処理は基本的にできず，複数の処理を組み合わせて実現するという思想になっています．

反面Perlのようなスクリプト言語は，そのようなコンピュータに近い部分は隠蔽し，プログラムを書く上で便利な様々な機能を始めから用意しておくことで，プログラムの密度を高くするという考えのものです．

つまりこれは，言語の「目的」による違いであり，どちらが優れているといった話のものではないわけです．このためプログラミングには，多種多様な言語の存在意義があるわけです．

●本書で紹介するCentOSのCUIコマンド ―5（16-18章）

コマンド	ツールの役割	ページ
chmod +x（ファイル名）	（ファイル名）実行属性を付加する ※スクリプトファイルなど。	291
gcc -c（Cプログラムファイル名）	（Cプログラムファイル名）.oというオブジェクトファイルを生成する	311
gcc（オブジェクトファイル1）.o（オブジェクトファイル2）.o -o（実行ファイル名）	（オブジェクトファイル1）.oと（オブジェクトファイル2）.oを結合して実行ファイルを生成[リンク]	312
zip（アーカイブファイル名）.zip Makefile *.h *.c	MakefileとCプログラムとヘッダファイルすべてを含めてアーカイブ化（圧縮）する	323

17

ソースコードを分割しよう

[C言語プログラミング編7]ー分割コンパイル

ここまでで書いたC言語のプログラムは，すべてひとつのファイルに記述していました．

しかし実際のプログラミングでは，そのようなことは稀です．プログラムのサイズが大きくなると，ひとつのファイルにすべての内容を収めることは見通しも悪くなり，管理も面倒になるためです．

そこでソースコードは，複数のファイルに分割されます．コンパイルもファイルごとに行われ，最終的にすべてのコンパイル結果を結合することで実行ファイルとします．このような手法は「分割コンパイル」と呼ばれます．

またサブルーチンを切り出して流用・再利用できるように部品化したものを「ライブラリ」と呼びます．

ここではC言語のソースコード分割と「分割コンパイル」の方法，ライブラリの作成を体験してみましょう．

17.1 ライブラリにしてみよう

ソースコード分割の題材としては，15章でとりあげたソートのプログラムの最終形（**リスト15.4**）を使いましょう．

ここで**リスト15.4**を**リスト17.1**として，もう一度掲載しておきます．

リスト17.1: sort.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: void exchange(int a[], int i)
005: {
006:         int x;
007:         x = a[i + 1];
008:         a[i + 1] = a[i];
009:         a[i] = x;
010: }
011:
012: int compare(int a[], int n)
013: {
```

```
014:         int changed;
015:         int i;
016:         changed = 0;
017:         for (i = 0; i < n - 1; i++) {
018:                 if (a[i] > a[i + 1]) {
019:                         exchange(a, i);
020:                         changed = 1;
021:                 }
022:         }
023:         return changed;
024: }
025:
026: int main(int argc, char *argv[])
027: {
028:         int a[5] = { 5, 3, 4, 2, 1 };
029:         int changed;
030:
031:         changed = 1;
032:         while (changed == 1) {
033:                 printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
034:                 changed = compare(a, 5);
035:         }
036:         exit(0);
037: }
```

リスト 17.1 では exchange(), compare(), main() という３つの関数を，sort.c というひとつのファイルにすべて記述しています．

これを「ライブラリ」として，複数のファイルに分割してみましょう．

17.1.1 ライブラリを作成してみよう

sort.c は git で管理されていますので，まずはそのディレクトリに入りましょう．

```
[user@localhost ~]$ cd git/sort
[user@localhost sort]$ make clean
rm -f sort
[user@localhost sort]$ ls
Makefile  sort.c
[user@localhost sort]$
```

ここに lib.c というファイルを作成し，sort.c にあった「exchange()」という関数をそちらに移動させてみましょう．

```
[user@localhost sort]$ nano lib.c
```

lib.c は**リスト 17.2**のようになります．これは，単純に関数 exchange() を移動しただけです．

リスト 17.2: lib.c

```
001: #include "lib.h"
002:
003: void exchange(int a[], int i)
004: {
005:         int x;
006:         x = a[i + 1];
007:         a[i + 1] = a[i];
008:         a[i] = x;
009: }
```

lib.c は関数 exchange() だけを，プログラムの本体である sort.c とは独立して定義しています．つまり lib.c は，「exchange」という「サブルーチン」を格納した「ライブラリ」と言うことができます．

また先頭には「#include "lib.h"」のような行があります．これは次に説明する「ヘッダファイル」を読み込み，関数 exchange() の定義に食い違いが無いかを確認するための記述です．

17.1.2 ヘッダファイルを作成してみよう

関数 exchange() を lib.c に移動しましたが，exchange() は sort.c の内部から利用します．

sort.c をコンパイルする際には，exchange() がどのような関数なのかといった情報が必要です．しかしこの情報は sort.c の中にはありません． exchange() の定義は sort.c の中からは削除されてしまうためです．

このため，関数 exchange() がどのような関数なのかを伝えるための「ヘッダファイル」というファイルを作成する必要があります．

まずヘッダファイルとして，lib.h というファイルを作成しましょう．

```
[user@localhost sort]$ nano lib.h
```

lib.h は**リスト 17.3**のような内容になります．

リスト 17.3: lib.h

```
001: #ifndef _LIB_H_INCLUDED_
002: #define _LIB_H_INCLUDED_
003:
004: void exchange(int a[], int i);
005:
006: #endif
```

ヘッダファイルには，関数exchange()の引数と戻り値の定義を記述します．このようなものは関数の「プロトタイプ宣言」と呼ばれます．

exchange() は sort.c から利用しますが，その本体の定義は lib.c にあります．つまり sort.c のコンパイル時には，exchange() の引数や戻り値の定義を知ることができません．

よってヘッダファイルを作成し，そこに exchange() の引数と戻り値の定義を記述しておきます．sort.c からは lib.h を参照することで，関数 exchange() の引数と戻り値がどのようなものかを知ることができます．

つまりヘッダファイルは，lib.cという「ライブラリ」を利用するための「仕様書」にあたるものです．lib.c がライブラリの本体，lib.h がライブラリの仕様書に相当するわけです．

17.1.3 ライブラリを利用しよう

sort.c からはライブラリを呼び出すように修正を加えます．

```
[user@localhost sort]$ nano sort.c
```

sort.c は**リスト 17.4**のようになります．

リスト 17.4: sort.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003: #include "lib.h"
004:
005: int compare(int a[], int n)
006: {
007:         int changed;
```

```
008:         int i;
009:         changed = 0;
010:         for (i = 0; i < n - 1; i++) {
011:                 if (a[i] > a[i + 1]) {
012:                         exchange(a, i);
013:                         changed = 1;
014:                 }
015:         }
016:         return changed;
017: }
018:
019: int main(int argc, char *argv[])
020: {
021:         int a[5] = { 5, 3, 4, 2, 1 };
022:         int changed;
023:
024:         changed = 1;
025:         while (changed == 1) {
026:                 printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
027:                 changed = compare(a, 5);
028:         }
029:         exit(0);
030: }
```

まず関数 exchange() の定義は lib.c に移動したため，sort.c からは削除しています．

また**リスト 17.4** では，３行目に以下の行が追加されています．

```
#include "lib.h"
```

これは lib.h を読み込むという意味です．「lib.h をインクルードする」などと呼ばれます．

lib.h には exchange() の引数と戻り値がどのようなものかが記述されています．さらに sort.c の内部では，exchange() を呼び出しています．この読み込みにより，sort.c の内部で exchange() が利用できるようになります．

git diff で sort.c の変更点を見てみましょう．

```
[user@localhost sort]$ git diff sort.c
diff --git a/sort/sort.c b/sort/sort.c
index b3e18df..bfebd18 100644
--- a/sort/sort.c
+++ b/sort/sort.c
@@ -1,13 +1,6 @@
 #include <stdio.h>
```

```
 #include <stdlib.h>
-
-void exchange(int a[], int i)
-{
-       int x;
-       x = a[i + 1];
-       a[i + 1] = a[i];
-       a[i] = x;
-}
+#include "lib.h"

 int compare(int a[], int n)
 {
[user@localhost sort]$
```

関数 exchange() の定義が削除され，lib.h のインクルードが追加されています．

17.1.4 分割コンパイルをしてみよう

これらのプログラムは，以下のようにして一気にコンパイルすることができます．

```
[user@localhost sort]$ gcc sort.c lib.c -o sort
```

しかしコンパイルエラーへの対処などを考えると，ファイルをひとつひとつ順番にコンパイルしたほうがいいでしょう．その方法を説明しましょう．

まず sort.c をコンパイルし，「オブジェクトファイル」というファイルを生成します．

```
[user@localhost sort]$ gcc -c sort.c
```

これで「sort.o」というオブジェクトファイルが生成されます．オブジェクトファイルは，実行ファイルを作成するための中間ファイルです．

```
[user@localhost sort]$ ls sort.*
sort.c  sort.o
[user@localhost sort]$
```

次に lib.c からオブジェクトファイルをコンパイルします．

```
[user@localhost sort]$ gcc -c lib.c
```

これで「lib.o」が生成されます．

```
[user@localhost sort]$ ls lib.*
lib.c  lib.h  lib.o
[user@localhost sort]$
```

さらにこれらの「sort.o」と「lib.o」を結合することで，実行ファイルを生成します．

```
[user@localhost sort]$ gcc sort.o lib.o -o sort
```

このような作業を「リンク」と呼びます．

そして実行ファイルの生成は，ソースコードのファイルからまずオブジェクトファイルを作成し，それらをリンクすることで行いました．このような手法が「分割コンパイル」と呼ばれるものです．

これで実行ファイルである「sort」が作成されました．実行してみましょう．

```
[user@localhost sort]$ ./sort
5 3 4 2 1
3 4 2 1 5
3 2 1 4 5
2 1 3 4 5
1 2 3 4 5
[user@localhost sort]$
```

無事にソートできています．

17.1.5 Makefileを分割コンパイルに対応させよう

「分割コンパイル」を体験してみましたが，このようにコンパイル作業を複数に分けて行うのではたいへん面倒です．

そこでコンパイル手順を Makefile にしてみましょう． Makefileは**リスト17.5**のようになります．

リスト17.5: Makefile

```
001: TARGET = sort
002: CFLAGS = -g
003:
004: all :          $(TARGET)
005:
006: $(TARGET) :    sort.o lib.o
```

```
007:                gcc $(CFLAGS) sort.o lib.o -o $(TARGET)
008:
009: sort.o :       lib.h sort.c
010:                gcc $(CFLAGS) -c sort.c
011:
012: lib.o :        lib.h lib.c
013:                gcc $(CFLAGS) -c lib.c
014:
015: clean :
016:                rm -f $(TARGET) sort.o lib.o
```

リスト17.5ではまず「all」というターゲットを定義してあります．make コマンドはそのまま実行すると先頭にあるターゲットが処理されますが，これによりコマンドライン引数無しで単に「make」が実行されると，all ターゲットが処理され，そこから連携して sort ターゲットが処理されることになります．

sort ターゲットは，sort.o と lib.o をリンクすることで，実行ファイル「sort」を生成します．依存するファイルは sort.o と lib.o で，それらはその直後でターゲットが定義されています．

sort.o と lib.o は分割コンパイルによりオブジェクトファイルを生成するようになっています．

さらに clean ターゲットは生成したファイルを削除し，ディレクトリの掃除を行います．

実際にコンパイルを行ってみましょう．

```
[user@localhost sort]$ make clean
rm -f sort sort.o lib.o
[user@localhost sort]$ make
gcc -g -c sort.c
gcc -g -c lib.c
gcc -g sort.o lib.o -o sort
[user@localhost sort]$
```

sort.c と lib.c がコンパイルされたあと，それらがリンクされ，実行ファイルが生成されています．

ここで例えば，lib.c に修正を加えたとします．そのような場合も make を実行するだけで，修正したファイルが検知され，必要な処理のみ行って実行ファイルが再生成されます．

```
[user@localhost sort]$ touch lib.c
[user@localhost sort]$ make
gcc -g -c lib.c
gcc -g sort.o lib.o -o sort
[user@localhost sort]$
```

lib.c はコンパイルされていますが，sort.c はコンパイルされていません． sort.c には変更は無いの

で，sort.o を再生成する必要は無いためです．

実行結果も確認しておきましょう．

```
[user@localhost sort]$ ./sort
5 3 4 2 1
3 4 2 1 5
3 2 1 4 5
2 1 3 4 5
1 2 3 4 5
[user@localhost sort]$
```

実行が確認できたので，作成したファイルをすべて git で登録しておきましょう．

```
[user@localhost sort]$ git add Makefile sort.c lib.c lib.h
[user@localhost sort]$ git commit Makefile sort.c lib.c lib.h
```

17.2 さらに分割を進めよう

変数の値の交換処理をライブラリ化しましたが，さらに進めて，比較処理もライブラリにしてみましょう．

compare.c と compare.h というファイルを作成し，sort.c の中にあった compare() という関数をライブラリ化します．移動の手順は lib.c/lib.h を作成したときと同じです．

17.2.1 比較処理をライブラリ化してみよう

まずは compare.c を作成して，sort.c の内部にあった compare() という関数をそちらに移動してみましょう．

```
[user@localhost sort]$ nano compare.c
```

compare.c の内容は**リスト 17.6**のようになります．関数 compare() が定義されているだけです．また lib.h をインクルードしていますが，これは内部から exchange() を呼び出しているためです．

リスト 17.6: compare.c

```
001: #include "lib.h"
002: #include "compare.h"
003:
004: int compare(int a[], int n)
005: {
006:         int changed;
007:         int i;
008:         changed = 0;
009:         for (i = 0; i < n - 1; i++) {
010:                 if (a[i] > a[i + 1]) {
011:                         exchange(a, i);
012:                         changed = 1;
013:                 }
014:         }
015:         return changed;
016: }
```

さらにヘッダファイルとして，compare.h を作成します．

```
[user@localhost sort]$ nano compare.h
```

compare.h の内容は**リスト 17.7**のようになります. 関数 compare() の引数と戻り値を記述しています.

リスト 17.7: compare.h

```
001: #ifndef _COMPARE_H_INCLUDED_
002: #define _COMPARE_H_INCLUDED_
003:
004: int compare(int a[], int n);
005:
006: #endif
```

17.2.2 比較処理のライブラリを利用しよう

次に sort.c を修正します.

```
[user@localhost sort]$ nano sort.c
```

sort.c は**リスト 17.8**のようになります.

リスト 17.8: sort.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003: #include "compare.h"
004:
005: int main(int argc, char *argv[])
006: {
007:         int a[5] = { 5, 3, 4, 2, 1 };
008:         int changed;
009:
010:         changed = 1;
011:         while (changed == 1) {
012:                 printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
013:                 changed = compare(a, 5);
014:         }
015:         exit(0);
016: }
```

sort.cからはcompare()の定義を削除し，かわりにcompare.hのインクルードを追加しています．また lib.h のインクルードは削除しました．これは compare() をcompare.c に移動し，exchange()の呼び出しが無くなったためです．

git diff で修正点を確認しておきましょう．

```
[user@localhost sort]$ git diff sort.c
diff --git a/sort/sort.c b/sort/sort.c
index bfebd18..e300f55 100644
--- a/sort/sort.c
+++ b/sort/sort.c
@@ -1,20 +1,6 @@
 #include <stdio.h>
 #include <stdlib.h>
-#include "lib.h"
-
-int compare(int a[], int n)
-{
-       int changed;
-       int i;
-       changed = 0;
-       for (i = 0; i < n - 1; i++) {
-               if (a[i] > a[i + 1]) {
-                       exchange(a, i);
-                       changed = 1;
-               }
-       }
-       return changed;
-}
+#include "compare.h"

 int main(int argc, char *argv[])
 {
[user@localhost sort]$
```


17.2.3 Makefileを修正しよう

さらに，Makefile も修正します．

```
[user@localhost sort]$ nano Makefile
```

Makefile は**リスト17.9**のようになります．

リスト 17.9: Makefile

```
001: TARGET = sort
002: CFLAGS = -g
003:
004: all :          $(TARGET)
005:
006: $(TARGET) :    sort.o compare.o lib.o
007:                gcc $(CFLAGS) sort.o compare.o lib.o -o $(TARGET)
008:
009: sort.o :       compare.h sort.c
010:                gcc $(CFLAGS) -c sort.c
011:
012: compare.o :    lib.h compare.h compare.c
013:                gcc $(CFLAGS) -c compare.c
014:
015: lib.o :        lib.h lib.c
016:                gcc $(CFLAGS) -c lib.c
017:
018: clean :
019:                rm -f $(TARGET) sort.o compare.o lib.o
```

Makefile には，compare.c のコンパイル処理を追加しています．

git diff で修正内容も確認しておきましょう．

```
[user@localhost sort]$ git diff Makefile
diff --git a/sort/Makefile b/sort/Makefile
index 7d73825..e124a31 100644
--- a/sort/Makefile
+++ b/sort/Makefile
@@ -3,14 +3,17 @@ CFLAGS = -g

 all :          $(TARGET)

-$(TARGET) :    sort.o lib.o
-               gcc $(CFLAGS) sort.o lib.o -o $(TARGET)
+$(TARGET) :    sort.o compare.o lib.o
+               gcc $(CFLAGS) sort.o compare.o lib.o -o $(TARGET)

-sort.o :       lib.h sort.c
+sort.o :       compare.h sort.c
                gcc $(CFLAGS) -c sort.c

+compare.o :    lib.h compare.h compare.c
+               gcc $(CFLAGS) -c compare.c
+
```

```
lib.o :                lib.h lib.c
               gcc $(CFLAGS) -c lib.c

clean :
-              rm -f $(TARGET) sort.o lib.o
+              rm -f $(TARGET) sort.o compare.o lib.o
[user@localhost sort]$
```

17.2.4 実行を確認しよう

これでソースコードの修正は完了です．

コンパイルと実行を確認してみましょう．

```
[user@localhost sort]$ make clean
rm -f sort sort.o compare.o lib.o
[user@localhost sort]$ make
gcc -g -c sort.c
gcc -g -c compare.c
gcc -g -c lib.c
gcc -g sort.o compare.o lib.o -o sort
[user@localhost sort]$ ./sort
5 3 4 2 1
3 4 2 1 5
3 2 1 4 5
2 1 3 4 5
1 2 3 4 5
[user@localhost sort]$
```

compare.c のコンパイルが行われています．ファイルは sort.c と lib.c と compare.c の３つに分割されましたが，実行にも問題は無いようです．

最後に，修正を加えたファイルをすべて git で登録しておきましょう．

```
[user@localhost sort]$ git add Makefile sort.c compare.c compare.h
[user@localhost sort]$ git commit Makefile sort.c compare.c compare.h
```

17.3 まとめ

分割コンパイルは make とうまく組み合わせることで，部分的に修正を加えたときのコンパイル作業を最小限にして，巨大なプログラムも効率良く開発していくことができます．

またソースコードを複数のファイルに分割することで見通しを良くし，管理しやすくします．

さらにソースコードを分割しておけば，特定の処理を流用する際に便利です．例えば今回作成した exchange() という関数を別のプログラムで流用したい場合にも，lib.c と lib.h のみを持っていけばいいことになります．このように分割コンパイルは，プログラムの「部品化」を進める上で非常に重要なわけです．

18

アーカイブにして配布しよう

[ツール編8]—zipコマンド

ここまでで様々なプログラムを作成してきました．

こうしたプログラムは，他人に渡したいときもあるかもしれません．また場合によっては，フリーソフトウェアとしてインターネット上で配布したい，という機会もあるかもしれません．

しかし渡したいファイルは複数あります．ソースコードもありますが，Makefile もあります．また使いかたを書いたドキュメントや，ライセンス文書などのファイルもあるかもしれません．

しかしこれらをひとつひとつ渡すことはたいへん面倒ですし，ファイルの渡し忘れなども起きそうです．

そのようなときには複数のファイルをひとつにまとめることで，受渡ししやすくすることができます．これを「アーカイブ」といいます．アーカイブを行うためのツールは「アーカイバ」と呼ばれます．

最後は本書で作成したソートプログラムを，アーカイブして配布する準備をしてみましょう．

18.1 ZIPフォーマットでアーカイブしてみよう

アーカイブには様々な方法がありますが，ここでは広く用いられている「ZIP」というフォーマットを紹介します．「ZIP」は「ジップ」と呼ばれます．

ZIPはアーカイブだけでなくファイルを圧縮する機能もあるため，ファイルのサイズを小さくして受渡しをすることも可能になります．

18.1.1 zipコマンドでアーカイブしてみよう

ソートプログラムはいくつかのファイルからできています．具体的には以下のファイルから構成されています．

- Makefile
- sort.c（本体のプログラム）
- compile.h（比較処理のヘッダファイル）
- compile.c（比較処理の本体）
- lib.h（交換処理のヘッダファイル）

- **lib.c（交換処理の本体）**

ひとまず，これらを確認してみましょう．

```
[user@localhost ~]$ cd git/sort
[user@localhost sort]$ make clean
rm -f sort sort.o compare.o lib.o
[user@localhost sort]$ ls
Makefile  compare.c  compare.h  lib.c  lib.h  sort.c
[user@localhost sort]$
```

これらのファイルを，ZIPでアーカイブしてみます．

ZIPでのアーカイブは，「zip」というコマンドで可能です．zipコマンドは，コマンドライン引数として最初にアーカイブ先のファイル名を指定し，さらにアーカイブするファイルを羅列します．

例えば以下のようにすれば，zipコマンドでソースコードをまとめることができます．

```
[user@localhost sort]$ zip sort.zip Makefile *.h *.c
  adding: Makefile (deflated 57%)
  adding: compare.h (deflated 25%)
  adding: lib.h (deflated 20%)
  adding: compare.c (deflated 36%)
  adding: lib.c (deflated 23%)
  adding: sort.c (deflated 37%)
[user@localhost sort]$
```

ソースコードはすべてを列挙するのは面倒なので，「*.h」と「*.c」のようにして2章で説明したワイルドカードを使って指定しましたが，「compare.h compare.c lib.h lib.c」のようにひとつひとつ指定しても構いません．

これで「sort.zip」というアーカイブが作成されました．なおZIPファイルの拡張子は「.zip」にするのが通例になっています．

18.1.2 unzipコマンドで展開してみよう

ファイルをアーカイブした後は，そのアーカイブの展開も試してみましょう．

まずアーカイブしたsort.zipをホームディレクトリに移動し，カレントディレクトリもそちらに移動します．

```
[user@localhost sort]$ mv sort.zip ../..
[user@localhost sort]$ cd ../..
```

さらに展開用のディレクトリを適当な名前で作成しましょう.

```
[user@localhost ~]$ mkdir tmp
[user@localhost ~]$ cd tmp
[user@localhost tmp]$
```

ここでは一時的なディレクトリとして,「tmp」というディレクトリを作成しました.「tmp」はtemporary の略で,「一時的な」という意味でファイル名やディレクトリ名などによく利用されます.

カレントディレクトリをtmpに移動し, sort.zip もそちらに移動します.

```
[user@localhost tmp]$ mv ../sort.zip .
[user@localhost tmp]$ ls
sort.zip
[user@localhost tmp]$
```

では, sort.zip を展開してみましょう. アーカイブしたファイルは「unzip」というコマンドで展開することができます.

以下のようにして, unzip コマンドを sort.zip を引数にして実行します.

```
[user@localhost tmp]$ unzip sort.zip
Archive:  sort.zip
  inflating: Makefile
  inflating: compare.h
  inflating: lib.h
  inflating: compare.c
  inflating: lib.c
  inflating: sort.c
[user@localhost tmp]$
```

無事に展開できたでしょうか. 確認してみましょう.

```
[user@localhost tmp]$ ls
Makefile  compare.c  compare.h  lib.c  lib.h  sort.c  sort.zip
[user@localhost tmp]$
```

ソートプログラムのソースコードとMakefileが出現しています. うまく展開できているようです.

18.1.3 コンパイルできることを確認しよう

展開はうまくできましたが，もしも足りないファイルなどがあるとコンパイルして実行することができないはずです．

makeして，実行ファイルを生成してみましょう．

```
[user@localhost tmp]$ make
gcc -g -c sort.c
gcc -g -c compare.c
gcc -g -c lib.c
gcc -g sort.o compare.o lib.o -o sort
[user@localhost tmp]$
```

実行もしてみます．

```
[user@localhost tmp]$ ./sort
5 3 4 2 1
3 4 2 1 5
3 2 1 4 5
2 1 3 4 5
1 2 3 4 5
[user@localhost tmp]$
```

問題ないようです．つまりソートプログラムを sort.zip というアーカイブにまとめ，それをコピーし，コピー先で展開し，オリジナルと同様にコンパイルして実行することができているわけです．

たとえばこの sort.zip をそのまま他人に渡すこともできますし，フリーソフトウェアとしてインターネット上で配布することもできるわけです．

18.2 配布形態に仕上げよう

zipコマンドでアーカイブし，unzipコマンドで展開することはできました．

しかし実際に配布するとしたら，コンパイル手順や利用方法を書いたドキュメントを添付しておくといいでしょう．このようなドキュメントは，慣例として「README」というファイルに書いておきます．

18.2.1 READMEを追加しよう

READMEを作成して登録するために，もう一度 git リポジトリのほうに戻ります．

```
[user@localhost tmp]$ cd ../git/sort
[user@localhost sort]$
```

nano でREADMEを作成します．

```
[user@localhost sort]$ nano README
```

ひとまずREADMEは**図18.1**のような内容で作成しました．概要，コンパイル方法，実行方法を記述してあります．

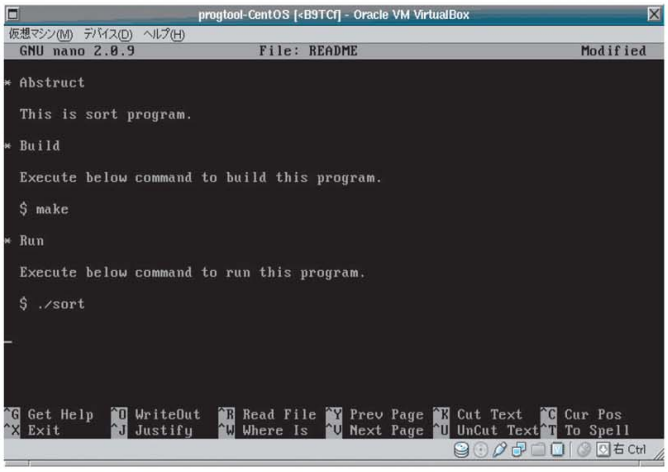


図18.1: READMEを作成する

README を記述したら Ctrl＋X で保存・終了します．

さらにこれも git で登録しておきましょう．

```
[user@localhost sort]$ git add README
[user@localhost sort]$ git commit README
```

18.2.2 Makefileを修正しよう

アーカイブの手順は Makefile にターゲットとして記述しておくことで，いつでも ZIP ファイルを生成することが可能になります．

そこで Makefile を**リスト 18.1** のように修正し，zip と sort.zip というターゲットを追加してみました．追加したのは，20行目の「rm -fR」の行以降です．

リスト 18.1: Makefile

```
001: TARGET = sort
002: CFLAGS = -g
003:
004: all :           $(TARGET)
005:
006: $(TARGET) :     sort.o compare.o lib.o
007:                 gcc $(CFLAGS) sort.o compare.o lib.o -o $(TARGET)
008:
009: sort.o :        compare.h sort.c
010:                 gcc $(CFLAGS) -c sort.c
011:
012: compare.o :     lib.h compare.h compare.c
013:                 gcc $(CFLAGS) -c compare.c
014:
015: lib.o :         lib.h lib.c
016:                 gcc $(CFLAGS) -c lib.c
017:
018: clean :
019:                 rm -f $(TARGET) sort.o compare.o lib.o
020:                 rm -fR sort-1.0 sort-1.0.zip
021:
022: zip :           sort-1.0.zip
023:
024: sort-1.0.zip :
025:                 rm -fR sort-1.0
026:                 mkdir -p sort-1.0
027:                 cp README Makefile *.h *.c sort-1.0
028:                 zip -r sort-1.0.zip sort-1.0
```

修正内容を git diff で確認してみましょう.

```
[user@localhost sort]$ git diff Makefile
diff --git a/sort/Makefile b/sort/Makefile
index e124a31..4451779 100644
--- a/sort/Makefile
+++ b/sort/Makefile
@@ -17,3 +17,12 @@ lib.o :              lib.h lib.c

 clean :
                rm -f $(TARGET) sort.o compare.o lib.o
+               rm -fR sort-1.0 sort-1.0.zip
+
+zip :          sort-1.0.zip
+
+sort-1.0.zip :
+               rm -fR sort-1.0
+               mkdir -p sort-1.0
+               cp README Makefile *.h *.c sort-1.0
+               zip -r sort-1.0.zip sort-1.0
[user@localhost sort]$
```

「zip」というターゲットを追加してあります．さらに zip ターゲットは，「sort-1.0.zip」というターゲットを実行します．

sort-1.0.zip ターゲットは，まず sort-1.0 というディレクトリを作成します．さらに必要なファイルを sort-1.0 にコピーし，zip コマンドでそれをアーカイブすることで，sort-1.0.zip というファイルを生成します．

zip コマンドに -r というオプションが付加されているのは，ディレクトリ内も含めて圧縮するという意味です．

なおアーカイブの名前は「sort-1.0」のようにしました．これは「バージョン1.0」という意味です．

18.2.3 アーカイブしてみよう

では，実際にアーカイブしてみましょう．

まず，make clean でディレクトリを掃除します．

```
[user@localhost sort]$ make clean
rm -f sort sort.o compare.o lib.o
rm -fR sort-1.0 sort-1.0.zip
[user@localhost sort]$
```

make zip を実行しましょう.

```
[user@localhost sort]$ make zip
rm -fR sort-1.0
mkdir -p sort-1.0
cp README Makefile *.h *.c sort-1.0
zip -r sort-1.0.zip sort-1.0
  adding: sort-1.0/ (stored 0%)
  adding: sort-1.0/lib.h (deflated 20%)
  adding: sort-1.0/Makefile (deflated 59%)
  adding: sort-1.0/README (deflated 37%)
  adding: sort-1.0/sort.c (deflated 37%)
  adding: sort-1.0/lib.c (deflated 23%)
  adding: sort-1.0/compare.c (deflated 36%)
  adding: sort-1.0/compare.h (deflated 25%)
[user@localhost sort]$
```

アーカイブは作成できたでしょうか．確認してみましょう.

```
[user@localhost sort]$ ls
Makefile  compare.c  lib.c  sort-1.0      sort.c
README    compare.h  lib.h  sort-1.0.zip
[user@localhost sort]$
```

sort-1.0.zip が生成されています.

18.2.4 アーカイブを確認しよう

生成した sort-1.0.zip をホームディレクトリに移動して，そちらで展開してみましょう.

```
[user@localhost sort]$ mv sort-1.0.zip ../..
[user@localhost sort]$ cd ../..
[user@localhost ~]$
```

unzip を実行して，sort-1.0.zip を展開します.

```
[user@localhost ~]$ unzip sort-1.0.zip
Archive:  sort-1.0.zip
   creating: sort-1.0/
  inflating: sort-1.0/lib.h
  inflating: sort-1.0/Makefile
  inflating: sort-1.0/README
  inflating: sort-1.0/sort.c
  inflating: sort-1.0/lib.c
  inflating: sort-1.0/compare.c
  inflating: sort-1.0/compare.h
[user@localhost ~]$
```

これで sort-1.0.zip が展開され，sort-1.0 ディレクトリが作成されています．

そこに移動して，ファイルを確認しましょう．

```
[user@localhost ~]$ cd sort-1.0
[user@localhost sort-1.0]$ ls
Makefile  README  compare.c  compare.h  lib.c  lib.h  sort.c
[user@localhost sort-1.0]$
```

ひととおりのファイルが展開できていることが確認できます．

コンパイルしてみましょう．

```
[user@localhost sort-1.0]$ make
gcc -g -c sort.c
gcc -g -c compare.c
gcc -g -c lib.c
gcc -g sort.o compare.o lib.o -o sort
[user@localhost sort-1.0]$
```

問題なくコンパイルできています．不足しているファイルなどは無いようです．

実行も確認しましょう．

```
[user@localhost sort-1.0]$ ./sort
5 3 4 2 1
3 4 2 1 5
3 2 1 4 5
2 1 3 4 5
1 2 3 4 5
[user@localhost sort-1.0]$
```

無事に実行までできました．

18.3 まとめ

本章で作成したソートプログラムは，バージョン1.0として sort-1.0.zip としてまとめました．これをソートプログラムの「バージョン1.0」として他人に渡したり，インターネット上で配布したりすることができるわけです．

なお実際に配布する際には，ライセンス文書も添付したほうがいいでしょう．ソフトウェアのライセンスについては様々な考慮すべき点があるため本書ではあえて言及しませんが，筆者としては「GPL」（GNU General Public License）というライセンスをお勧めしておきます． GPLについてはインターネット上で検索すると，いろいろ知ることができるでしょう．

おわりに

プログラミングもツールの操作も，修得に必要なことは，まずは実践です．

チュートリアル的な説明に沿って手を動かしてみることで得られることは，意外に多いものです．我々の知識は理論による理解ではなく，体験によって構成される部分が多くあるように思います．プログラミング教育も，アルゴリズムを考えることなどが重要視されがちですが，現実のプログラマは意外なほどに経験則によってプログラムを書いているものです．

もちろん理解することも重要なのですが，新しいことを身につけようとするときに，まず大切なことは「体験してみること」だと筆者は思っています．英語を初めて学ぶ際にまず必要なことは，文法書を通して読むことではなく，「This is a pen.」などと声に出して言ってみることなのです．また体験は理解のためだけでなく，モチベーションの維持や興味の促進といった二次的要素も多くあるでしょう．

そしてそのような経験を通して身についた知識は，今後プログラミング言語やツール類を自身で選択しなければならなくなったときに，きっと役に立つことでしょう．一番重要なことは，「自身で選択することができるようになること」です．

プログラミングは単にプログラムを書くだけの作業ではなく，こうしたツール類を選び，使いこなすための学習の連続です．良いプログラマは良いツールを選んで使うものですが，これはプログラマに限らない話かもしれません．読者の方々がそうしたツール類を選ぶ上での指南に本書がなることができれば幸いに思います．

本書は読者の方々がフリーソフトウェアを作成したり，勉強会に行ってみたりするためのきっかけになればと思い執筆しました．

プログラミングでもツール利用でもそうなのですが，上達のための一番の近道は，実際に手を動かしていじってみることです．しかし個人で勉強していると，なかなかそのためのモチベーションを維持することも難しいものです．

そのために筆者がお勧めする方法として，「勉強会に参加してみる」というものがあります．

近年，IT勉強会というものが各地で行われています．多くは有志で行われているボランティアベースの勉強会です．インターネットで検索すると，様々なものが見つかるでしょう．

そして近場で興味をひかれるものが見つかったら，ぜひ参加してみましょう．慣れてきたら，発表もしてみましょう．自身からの情報発信をすることで，モチベーションはさらに高まることでしょう．
本書を執筆した目的には，そうした勉強会に参加する際にそもそもプログラミングやCUIでの操作の経験が無いために本題に入る以前のところで立ちいかないというときのチュートリアルを提供したい，というものもありました．本書の操作がひととおりできれば，CUIによる操作はたとえ十分とは言えなくとも，抵抗感は払拭できるかと思います．

筆者も様々な勉強会に参加しています．ぜひ，どこかの勉強会でお会いしましょう．

Index　索引

■記号

■英字

■あ～た行

■は～ら行

著者略歴
坂井弘亮（さかい ひろあき）
幼少の頃よりプログラミングに親しみ，趣味での組込みOS自作，アセンブラ解析，イベントへの出展やセミナーでの発表などで活動中。代表的な著書は『12ステップで作る 組込みOS自作入門』（カットシステム）、『31バイトでつくるアセンブラプログラミング -アセンブラ短歌の世界-』、『0と1のコンピュータ世界 バイナリで遊ぼう!』（マイナビ、共著） セキュリティ＆プログラミングキャンプ（現セキュリティ・キャンプ）講師（2010年～）技術士（情報工学部門）。

本書の内容に関する質問は、下記のメールアドレスまで、お送りください。電話によるご質問、本書の内容以外についてのご質問についてはお答えできませんので、あらかじめご了承ください。
メールアドレス pc-books@mynavi.jp
本書の追加・正誤情報サイト　http://book.mynavi.jp/support/pc/5149/

開発ツールを使って学ぶ！ C言語プログラミング

2014年8月25日　初版第1刷発行・電子版 Ver1.00

著　者	坂井弘亮
発行者	中川信行
発行所	株式会社 マイナビ 〒100-0003 東京都千代田区一ツ橋1-1-1　パレスサイドビル TEL：03-6267-4431（編集） URL：http://book.mynavi.jp
カバーデザイン	結城亨（SelfScript）
制作	企画室ミクロ
編集担当	山口正樹

Contents

開発ツールを使って学ぶ!
C言語プログラミング

坂井弘亮 著
Hiroaki Sakai

開発体験。
プロが使うツールでプログラミング・マスター!

Linux操作・エディタ
diff/patchによるパッチ作成
gitによるコード管理
GDBデバッガ

基本からアルゴリズム入門
分割コンパイル
アーカイブ配布まで!

書籍サポートサイトより、学習用開発環境をダウンロード提供

マイナビ