プログラミング演習ガイダンス資料

秋田県立大学 システム科学技術学部
電子情報システム学科

2008 年度版

目次

第 I 部

プログラミング演習 履修のてびき

第 1 章

プログラミング演習実施要領

1.1 授業目標

コンピュータは現代社会では必要不可欠なツールである。特にエンジニアは問題解決のために自らプログラムを作成する能力が必要とされる。本演習では、C 言語を用いたプログラミングを通じて、プログラムの標準的な書き方を習得し、プログラム作成能力を養う。

1.2 授業計画

本演習は以下のような予定で行う。括弧内に対応するテキストの章番号を示した。

1 回　プログラミング入門 (1〜3 章)
2 回　基本的なCの規則 (4 章)
3 回　簡単なデータ入出力 (5 章)
4 回　簡単な計算 (6 章)
5 回　配列 (11 章)
6 回　条件分岐 (8 章)
7 回　条件による繰り返し (9 章)
8 回　回数による繰り返し (10 章)
9 回　関数 1（処理の分解）(12 章)
10 回　関数 2（関数の利用と変数のスコープ）(12 章)
11 回　ポインタの基礎（アドレスとポインタ）(14 章)
12 回　ポインタの応用（ポインタと関数，配列）(14 章)
13 回　構造体 (16 章)
14 回　演習 （セメスタ課題に関する質問等受付）

1.3 テキスト・参考書

テキスト

吉村賢治著:「C 言語によるプログラミング入門」, 昭晃堂, 2,310 円 （税込）

参考書

1. B. W. カーニハン, D. M. リッチー 著, 石田晴久 訳: 「プログラミング言語 C 第 2 版 ANSI 規格準拠」, 共立出版, 2,940 円（税込）
   (C 言語をより詳しく知りたい人に)
2. (株) アンク 著 :「C の絵本 – C 言語が好きになる 9 つの扉」, 翔泳社, 1,449 円（税込)
   (C 言語のわかりやすい参考書)
3. B. W. カーニハン 著, 木村泉 訳: 「プログラム書法 第 2 版」, 共立出版, 3,150 円（税込）
   (良いプログラムを書きたい人に)
4. 山口 和紀, 古瀬 一隆 著: 「新 The UNIX Super Text 上 改訂増補版」, 技術評論社, 3,654（税込）
   (UNIX をより詳しく知りたい人に)
5. D. キャメロン 著: 「入門 GNU Emacs 第 3 版」, オライリー・ジャパン, 3,990 円（税込）
   (Emacs を活用したい人に)
6. R. メクレンバーグ 著: 「GNU Make 第 3 版」, オライリージャパン, 2,940 円（税込）
   (make を活用したい人に)

など。他にも良い参考書は多数あるので、図書館等で調べると良い。

1.4　演習環境

計算機:　IBM PC
オペレーティングシステム:　Linux (2.4.27)
言語:　ANSI C
コンパイラ:　gcc (3.3.5)

1.5　演習室使用時間

- 演習室 (GI-201) は**平日 8 時〜21 時** (他の授業がない場合のみ) を使用可とする。
- 長期休暇中の使用可能時間については別途通知する。
- 使用可能時間以外の時間帯、休日等に演習室を使用する場合には、**コンピュータ実習室使用許可申請書**をコピーし、必要事項を記入のうえ提出すること (付録 B 参照)。

1.6　演習の進め方

本演習では、毎回新しいプログラミング技法の習得を目指す。各演習では、新しい技法を学習した後に、その技法に関連した課題を行い、プログラムやレポートとして提出する。

1.7　課題

課題には、**基本課題**、**応用課題**、**セメスタ課題**の 3 種類がある。基本課題と応用課題は毎回の授業で与えられる課題であり、セメスタ課題はセメスタを通じて実施する課題である。

基本課題: **各演習時間内**に提出する課題。

応用課題: **次週の演習開始前**までに提出する課題。

セメスタ課題: セメスタを通じて 2 つ与えられる課題。(提出期限はスケジュールを参照のこと。セメスタ課題は、他の課題と異なり**レポート**も同時に提出しなければならない。)

課題は、初回からその回までの内容を全て習得したものとして与える。したがって、理解が不十分なまま演習を終了すると、後々まで影響するので注意すること。たとえプログラムが実行可能であったとしても、十分な理解がないままプログラムを提出しないこと。第 1 回の課題以外はすべてプログラムの作成である。課題は特に定められた回およびセメスタ課題のレポートを除いてすべて電子的に提出する (ガイダンス資料 5 章参照)。なお、プログラムは本演習で定められたスタイルに沿って作成すること (第 II 部 プログラミング演習スタイル規則参照)。

課題の提出締切後、その課題の再提出締切以前であれば**受講者各自の判断で**任意に再提出できる。判断の材料として、プログラミング演習の Web ページの採点結果や担当の教官から与えられるヒントを利用できる (ガイダンス資料 5 章参照)。

1.8 課題提出スケジュール

基本課題、応用課題の提出に関係するスケジュールは以下の通りとする。

基本課題 本提出締切	演習日当日	17:40 (授業終了時)
基本課題 採点結果通知	基本課題 本提出締切 5 日後	9:00
応用課題 本提出締切	次回演習日	14:30 (演習開始前)
応用課題 採点結果通知	応用課題 本提出締切 5 日後	9:00
基本・応用課題再提出締切	次々回演習日	14:30 (演習開始前)

ただし、締切等の該当日が休日の場合は、翌日に繰り越す。また、採点結果の通知日は変更する場合がある。変更がある場合には、Web ページ等で通知する。

例として、第 1 週に関する各クラスのスケジュールを示す。

	木曜クラス	金曜クラス
基本課題 本提出締切	4 月 17 日	4 月 18 日
基本課題 採点結果通知	4 月 22 日	4 月 23 日
応用課題 本提出締切	4 月 24 日	4 月 25 日
応用課題 採点結果通知	4 月 30 日	4 月 30 日
基本・応用課題 再提出締切	5 月 2 日	5 月 2 日

セメスタ課題の提出に関係するスケジュールは以下の通りとする。提出締切時刻はいずれも 21 時とする。

	木曜クラス	金曜クラス
セメスタ課題 1 本提出締切	6 月 26 日	6 月 27 日
セメスタ課題 1 採点結果通知	7 月 3 日	7 月 4 日
セメスタ課題 1 再提出締切	7 月 17 日	7 月 18 日
セメスタ課題 2 本提出締切	7 月 10 日	7 月 11 日
セメスタ課題 2 採点結果通知	7 月 17 日	7 月 18 日
セメスタ課題 2 再提出締切	7 月 24 日	7 月 25 日

1.9 評価

評価は提出された課題により行う。各課題の評価結果は Web 上で受講者に対し示される (ガイダンス資料 5 章参照)。Web 上に表示される各課題の採点結果は

○　(80〜100 点)
△　(60〜79 点)
×　(0〜59 点)

の 3 段階である。[*1]

課題の重み配分は次の通り。この重みに各課題の採点結果を掛けた点数が本演習全体の評点 (100 点満点) となる。

課題の種類	満点	1 課題の重み	回数	計
基本課題	100	0.04	13 回	52
応用課題	100	0.02	13 回	26
セメスタ課題 1	100	0.11	1 回	11
セメスタ課題 2	100	0.11	1 回	11
合計				100

以下の基準を全て満たす場合のみ成績評価の対象（**条件をひとつでも満たさない者は自動的に不合格**）とする。

- 全ての演習（第 1 回〜第 14 回）に全て出席していること
- 全ての基本課題（1〜13）をそれぞれの本提出締切日時までに提出していること
- セメスタ課題 1 を本提出締切日時までに提出していること
- セメスタ課題 2 を本提出締切日時までに提出していること

※病気などでやむを得ず欠席・提出の遅延が発生する場合は、事由を証明する書類（診断書等）を提出し担当教員の指示を仰ぐこと。

[*1] 1 回の課題が複数の枝問に分割される場合には、枝問の点数の合計が課題の点数となる。

第 2 章

Unix 入門

2.1 Unix とは？

Unix (ユニックスと読みます) は、OS (オペレーティングシステム) の一種です。OS は「基本ソフトウェア」とも呼ばれ、コンピュータの上で様々なソフトウェアを動作させるために不可欠なソフトウェアです。Windows や MacOS も OS の一種です。本演習で使用するコンピュータに入っている **Linux** (リナックス、リヌクス) は Unix の一種 (亜種) の OS です。

2.2 端末とコマンド

Unix では多くの作業をマウスではなく、キーボードからの入力で行えるようになっています。そのために、**端末ウィンドウ** (仮想端末とも呼びます) が用意されています。端末ウィンドウは、GNOME **端末** (グノーム端末) と呼ばれるプログラムを起動することで表示できます。以後、端末ウィンドウを表示することを「端末を開く」と表現します。

端末を開くとウィンドウのほとんどの部分は空白で、左上に b00b0yy@txx:~$ のような記号が表示されている状態になります。この記号を「**コマンドプロンプト**」または単に「**プロンプト**」と呼びます。プロンプトの後には、黒い四角で**カーソル**が示されます。

```
b00b0yy@txx:~$ ■
```

プロンプトの後に文字列を入力し [Enter] キーを押すと、Unix はその文字列に対応したプログラムを実行して、(必要ならば) 結果を表示します。例えば、*date* と入力すると

```
b00b0yy@txx:~$ date
2008 年  4 月  1 日  火曜日  12:31:33 JST
b00b0yy@txx:~$ ■
```

などと表示されます。

上記の「date」のように、Unix に対する命令を表す文字列を**コマンド**と呼びます。例えば、date は今日の日付と時刻を表示するコマンドです。多くのコマンドは、あらかじめ用意されたコマンドと同じ名前のプログラムを実行して、その結果を表示します。プログラムが実行を終了すると再びプロンプトが表示されて、次のコマンドを入力できる状態になります。[*1]

[*1] 通常は、コマンドにより実行されたプログラムが終了しない限り、次のコマンドを入力することはできません。

実は、端末ウィンドウの中では「**シェル**」と呼ばれるプログラムが常に動いています。プロンプトを表示したり、コマンドの文字列に対応するプログラムを実行したりという作業は、全部シェルが行っています。このシェルというプログラムの使い方を覚えることで、コマンドの入力が容易に行えるようになっています。シェルの上手な使い方は、後で説明します。

2.3 パスワードの変更

Unix システムはユーザ名とパスワードだけで本人の認証を行いますので、特にパスワードの取り扱いは慎重にしなければなりません。パスワードが他人に漏れると、自分になりすましてシステムにログインできてしまうばかりか、システム全体を危険に落としいれてしまう可能性もあります。

このようなことにならないために初期パスワードを、以下の手順で必ず変更してください。

端末ウィンドウ上 (プロンプトの後のカーソルが黒くなっている事を確認してください) で

```
b00b0yy@txx:~$ yppasswd
```

と入力し (「b00b0yy@txx:~$」の部分は入力しません)、リターンキーを押します。指示に従って元のパスワード (初回の場合は初期パスワード) を 1 回入力し、新しいパスワードを 2 回入力します。

```
b00b0yy@txx:~$ yppasswd
Changing NIS account information for b00b0yy on cs.ec.honjyo.akita-pu.ac.jp.
Please enter old password:  (現在のパスワードを入力)
Changing NIS password for b00b0yy on cs.ec.honjyo.akita-pu.ac.jp.
Please enter new password:  (新しいパスワードを入力)
Please retype new password:  (再度、新しいパスワードを入力)
The NIS password has been changend on cs.ec.honjyo.akita-pu.ac.jp.
```

入力中の文字は実際には**一切表示されません**ので、慎重に入力してください。

新しいパスワードは以下の規則に従うように作成してください。

- 英文字の**大文字・小文字・記号・数字**を**必ず混ぜて**ください。
- 6 文字以上、8 文字以下で作成してください。
- ユーザ名と同一の文字列を含まないようにしてください。
- 氏名、生年月日、車のナンバー、電話番号、誕生日などを含まないようにしてください。

パスワードでは、大文字と小文字は区別されるので注意しましょう。テンキーは、NumLock キーの状態によって動作が異なりますので、使わない方が無難です。新しいパスワードとしてふさわしくない文字列 (簡単すぎるなど) を入力した場合には、警告されます。パスワードを正しく変更できた場合にのみ

```
The NIS password has been changend on cs.ec.honjyo.akita-pu.ac.jp.
```

のように表示されますので、このように表示されなかった場合にはもう一度最初からやりなおしてください。この手順で何度でもパスワードを変更することができます。

2.4 カレントディレクトリ

Unix ではファイルを分類し、まとめて管理するために Windows 等の場合と同じように**ディレクトリ** (Windows や MacOS の場合は「フォルダ」とも呼ばれます) を使います。それぞれの実行中のプログラムには、「今いるディレクトリ = **カレントディレクトリ**」があります。シェルもプログラムの一種なので、カレントディレクトリがあります。シェルのカレントディレクトリを調べるコマンドは「pwd」です。例えば

```
b00b0yy@txx:~$ pwd
/home/student/b00/b00b0yy
```

これは、シェルのカレントディレクトリが、/home/student/b00/b00b0yy というディレクトリであることを示しています。Unix では Windows と異なりディレクトリの階層の区切りは¥ではなく、「/ (スラッシュ)」で表します。また、ドライブの概念がないので先頭にドライブを表す C: などの文字は付きません。

2.4.1 ディレクトリの作成 (mkdir)

さて、カレントディレクトリの下 (中) に、新しいディレクトリを作成してみましょう。カレントディレクトリが /home/student/b00/b00b0yy のときに

```
b00b0yy@txx:~$ mkdir sample
```

と入力すると、/home/student/b00/b00b0yy/sample というディレクトリが作成されます。ここで、「mkdir」は**ディレクトリを作成する**という機能を持ったプログラムの名前ですが、その後ろの「sample」は作成するディレクトリの名前です。このようにコマンドでは、プログラム名の後にプログラムに付加的に与える情報を書くことができます。これを、コマンドの**引数 (ひきすう)** と呼びます。この例では、「mkdir というプログラムに、引数で与えられた「sample」というディレクトリを作りなさい」と指示していることになります。

2.4.2 ディレクトリ内容の表示 (ls)

カレントディレクトリの内容を調べるには、「ls」コマンドを用います。

```
b00b0yy@txx:~$ ls
Desktop/    sample/
```

ls は引数なしで用いると、**カレントディレクトリにあるファイルやディレクトリの一覧を表示**します。本演習で使用している ls プログラムは、ディレクトリと他のファイルを色で区別して表示します。また、ディレクトリの場合は名前の後に「/」(スラッシュ) を表示します。上の例では、カレントディレクトリには「Desktop」というディレクトリと「sample」というディレクトリがあり、他にファイルはありません。

ls に引数を指定すると、**引数で与えられたディレクトリにあるファイルやディレクトリの一覧を表示**します。たとえば、現在の状態では「sample」というディレクトリの内容は空ですので

```
b00b0yy@txx:~$ ls sample
```

のように何もファイルやディレクトリがないことがわかります。

2.4.3 カレントディレクトリの変更 (cd)

つぎに、シェルのカレントディレクトリを新しく作成したディレクトリ「sample」にしてみましょう。**カレントディレクトリ**の変更は「cd」というコマンドで行います。

```
b00b0yy@txx:~$ cd sample
```

とすると、シェルのカレントディレクトリが /home/student/b00/b00b0yy/sample になります。もう一度、pwd コマンドを実行してみましょう。

```
b00b0yy@txx:~/sample$ pwd
/home/student/b00/b00b0yy/sample
```

このように、カレントディレクトリが先程とは変わっているはずです。

ところで、最初はプロンプトが「b00b0yy@txx:~$」だけだったのに、カレントディレクトリが「sample」に変わるとプロンプトも「b00b0yy@txx:~/sample$」に変化したことに気がつきましたか？本演習で使っているシェルでは、プロンプト内にユーザ名, ホスト名[*2], カレントディレクトリを表示するようになっていて、pwd コマンドを使わなくてもカレントディレクトリがわかるようになっています。

ただし、カレントディレクトリのうち/home/student/b00/b00b0yy の部分は、「~」の 1 文字に置き換えられています。この「~」に置き換えられた部分を「**ホームディレクトリ**」と呼びます。新しく端末を開いた場合には、カレントディレクトリはホームディレクトリになっています。ですから、端末を開いた直後のプロンプトは「b00b0yy@txx:~$」だったのです。

今いるディレクトリを出てその上のディレクトリに戻るには

```
b00b0yy@txx:~/sample$ cd ..
b00b0yy@txx:~$ pwd
/home/student/b00/b00b0yy
```

のように、cd の引数に[*3]「..」という文字列 (ピリオド 2 つ) を与えます。[*4]

また、cd を何も引数を付けずに用いると、どんな場合でもホームディレクトリに戻ることができます。

[*2] Unix では、コンピュータの名前のことを「ホスト名」といいます。

[*3] cd と引数「..」の間には空白が必要です

[*4] 「..」(ピリオド二つ) は「**カレントディレクトリを中に含んでいるようなディレクトリ**」という意味です。また「.」(ピリオド一つ) で**カレントディレクトリ**を表します。

2.5 ファイル

2.5.1 標準出力とリダイレクト

ファイルを作成するにはいろいろな方法があります。一般的には、エディタを使用してファイルを作成することが多いでしょう。

また、コマンドが端末ウィンドウ上に行う表示 (**標準出力**といいます) の内容をファイルに書き出すこともできます。例えば、date コマンドで表示される結果を 「a」という名前のファイルに出力しましょう。

```
b00b0yy@txx:~$ date > a
```

「>」 は本来端末ウィンドウ上に表示される標準出力の内容をファイルに出力することを意味し、「**標準出力のリダイレクト**」と呼ばれます。

その後、ls コマンドで「a」というファイルができているか確認しましょう。

```
b00b0yy@txx:~$ ls
Desktop/    a     sample/
```

ディレクトリ「sample」の他に今作成したファイル「a」が表示されています。[*5]

2.5.2 ファイル内容の表示

ファイルの内容を確認したい場合は、「lv」というコマンドを使うと簡単に内容を見ることができます。

```
b00b0yy@txx:~$ lv  a
=========================================
ファイル「a」の内容が表示される
=========================================
```

lv でファイルの中身を見ている時に、スペースキーを押すと今見ている部分の次のページを表示します。また、「*b*」を押すと一つ前のページを見ることができます。見終わったら、「*q*」を押すと、lv は終了し、シェルのプロンプトが表示されます。

2.5.3 ファイルのコピー

ファイルをコピーするには、「cp」というコマンドを用います。ファイル「a」をファイル「b」にコピーするには

```
b00b0yy@txx:~$ cp  a  b
b00b0yy@txx:~$ ls
Desktop/    a     b     sample/
```

[*5] 「a」はディレクトリではなく普通のファイルなので、後に「/」は付きません。

のようにします。ファイル「b」が新たにできているのがわかります。lv を用いて「a」 と 「b」 の内容が同じであることを確認してください。

cp はコピー先にファイル名を指定する他に、ディレクトリ名を指定することもできます。

ファイル「a」をディレクトリ「sample」の中にコピーするには

```
b00b0yy@txx:~$ cp a sample
b00b0yy@txx:~$ ls
Desktop/    a    b    sample/
b00b0yy@txx:~$ ls sample
a
```

のようにします。

2.5.4 ファイル・ディレクトリの移動

ファイルの名前を変えたり、ファイルを移動したりするのには、「mv」というコマンドを使います。使いかたは、cp とほぼ同じです。

ファイル「b」を「d」という名前に変えるのには

```
b00b0yy@txx:~$ mv b d
b00b0yy@txx:~$ ls
Desktop/    a    d    sample/
```

のようにします。[*6]

2.5.5 ファイルの消去

ファイルを消すには、「rm」というコマンドを使います。

ディレクトリ「sample」の下のファイル「a」を消すには

```
b00b0yy@txx:~$ cd sample
b00b0yy@txx:~/sample$ ls
a
b00b0yy@txx:~/sample$ rm a
b00b0yy@txx:~/sample$ ls

```

のようにします。

rm や mv、cp を使うときは、必要なファイルを消したり、上書きしてしまわないよう十分注意してください。一度消えたファイルの内容は、原則的に二度と復活できません。

[*6] mv でも cp と同じように移動先としてディレクトリ名を指定することもできます。

2.6 シェルの便利な機能

2.6.1 ヒストリ機能

何度も同じ作業を繰り返すときなどには、前に入力したコマンドをもう一度プロンプトの後に表示できれば便利です。このようなときは、カーソルキーの [↑] や “C-p” (C は [Ctrl] キーを表す。したがって “C-p” は [Ctrl] キーを押しながら [p] を押す) を押すことで、以前入力したコマンドを表示することができます。もう一度、キーを押すともう一つ前のコマンドが表示され、キーを押す度に以前に入力したコマンドにさかのぼって表示します。入力したいコマンドが表示されたらそのまま [Enter] キーを押せば、そのコマンドを入力したのと同じ結果が得られます。この機能をシェルの**ヒストリ (履歴)** と呼びます。

2.6.2 行内編集機能

コマンドを入力していて途中の文字を間違った時には、カーソルキーの [→]、[←] や “C-b”, “C-f”, “C-a”, “C-e” などを使ってカーソルを修正したい場所に移動して、文字を修正することができます。修正の際には、Emacs (エディタの一つ、3 章参照) の文字列編集の機能とほぼ同様の機能が使えます。この機能をシェルの**行内編集機能**と呼びます。

2.6.3 補完機能

コマンドのプログラム名や引数に与えるファイル名などは、最初の何文字かを入力して [Tab] キーを押すと残りの部分を自動的に補完して入力してくれます。もし、残りの部分が何種類かある場合には、1 度 [Tab] キーを押しただけでは何も起きませんが、もう 1 度 [Tab] キーを押すとその文字列で始まるプログラム名やファイル名の一覧を表示してくれます。これをシェルの**補完機能**と呼びます。

2.7 コマンドの使い方を調べる (man)

ここまで、mkdir, cd, ls など様々なコマンドを用いましたが、これらのコマンドのより詳しい使い方は、「man」というコマンドを用いることで調べることができます。例えば、mkdir の使い方を表示するには

```
b00b0yy@txx:~$ man mkdir
========================================
  mkdir の使い方が表示される
========================================
```

とします。ここで表示されるのは、そのコマンドついての説明が書いてあるファイルの内容で、lv を用いて表示しています。したがって、ページの表示操作には lv の規則が適用されます。ls や cp には便利な使い方 (ディレクトリまるごとコピーするなど) がありますので、man でチェックしてみると良いでしょう。

Unix で使えるコマンドの一覧は、コマンドプロンプトが出ているとき (何も入力していないとき) に、[Tab] キーを 2 回押すことで見ることができます。ただし、本演習で使用するコンピュータでも、使えるコマンド (プログラム) の数は 1000 を越えます (それでも少ない方です)。これを全部調べるのは不可能ですので、上で紹介したような重要ないくつかのコマンドを覚えて、それを組み合わせて作業するようにすると良いでしょう。

第 3 章

エディタ入門 (Emacs)

3.1 Emacs とは？

Emacs (「イーマックス」と読みます) は、プログラム開発の現場で現在最も普及しているエディタの一つです。「エディタ」というのは、ファイルを編集するために使うプログラムの事で、Windows の「メモ帳」を高度にしたものと考えると良いでしょう。

Emacs には

- 全ての操作をキーボードから手を離さずに行える
- 強力なカスタマイズ機能がある
- プログラムのソースファイルの入力を助ける様々な機能がある

などの特徴があります。

授業では、Emacs のおおまかな使い方しか説明しませんが、プログラミングの際に便利な機能がたくさんありますので、興味のある人は参考書などを参照すると良いでしょう。

3.2 Emacs の起動

プログラミングを行っている際には、通常、端末プログラム上から Emacs を起動します。

```
b00b0yy@txx:~$ emacs &
```

Emacs を起動するためのコマンドは、全て小文字で「emacs」です。

- 最後に「&」を付けるのを忘れないようにしてください。

「&」を付け忘れると、Emacs を終了するまで次のコマンドを入力できなくなります

この起動方法を用いるときには、編集したいファイルをコマンドの引数に指定することもできます。「emacs」の後に、スペースで区切って編集したいファイル名を入力します。指定したファイルがカレントディレクトリになければ指定した名前のファイルを新たに作成し、あればそのファイルを開きます。例えば、「hello.c」という名前のファイルを編集したい時には、以下のようにします。

```
b00b0yy@txx:~$ emacs hello.c &
```

3.3　Emacs の終了

Emacs を終了するには、コントロールキー ([Ctrl] キー) を押しながら、“x” のキーを押します。その後、コントロールキーを押しながら、“c” のキーを押します。コントロールキーを押しっぱなしにしたまま、“xc” と順に押しても OK です。この操作を、“C-x C-c” と書きます。[*1]

もし、変更後に保存していなかった場合は

```
Save file /home/(ファイル名)?  (y, n, !, ., q, C-r or C-h)
```

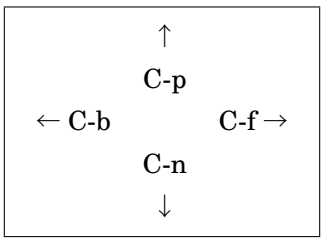
と最下行に表示されますので、「y」を入力するとファイルを保存して終了します。「n」を入力すると

```
Modified buffers exist; exit anyway?  (yes or no)
```

と表示されますので、「no」と答えると Emacs を終了せずに編集画面に戻ります。「yes」と答えると (3 文字全部打ってください) ファイルに保存せずに終了します。

3.4　カーソルの移動

矢印キーでカーソル移動しますが、以下のキーでも移動できます。



さらに、“C-a” で行の左端に、“C-e” で行の右端に移動します。

3.5　文字の入力

アルファベットの場合は、キーを押せばそのまま入力されます。

日本語の場合は “半角/全角” を押すと、一番下から 2 行目の黒い帯の行 (モード行) の左側に [あ] と表示されます。この状態の時は、ローマ字仮名漢字変換入力が可能になっています。ローマ字を入力すればかなに変換されますので、スペースを押すと文節ごとに分かれて漢字に変換されます。このときモード行の [あ] は [漢字] になっています。もう一度スペースキーを押すと候補の一覧がモード行に表示されます。カーソルを移動してリターンキーを押すとその漢字に変換されます。次の文節に移りたいときは “C-f”, 前の文節に戻りたいときは “C-b” を押します。また、文節を伸ばしたいときは “C-o”, 縮めたいときは “C-i” を押します。確定はリターンキーです。日本語入力をやめたいときは、もう一度 “半角/全角” を押すと [あ] の表示が消え、アルファベット入力モードに戻ります。

[*1] 別の方法として、上段のメニューの [Files] から [Exit Emacs] の項目をマウスで選んでも終了できます。さらに簡単な方法は、ウィンドウの左上の、[x] 印のボタンをクリックする方法です。

3.6 ファイルを開く/ファイルに保存

ファイルを開くには、“C-x C-f” と押します。ウィンドウの一番下の行 (ミニバッファ) に、

```
Find file:  ~/
```

と表示されるので、ファイル名を入力します。

ファイル名がわからなくなった場合には、「Find file: ~/」と表示されている状態でとりあえずスペースキーを押してみてください。ファイルの一覧が表示されます。その中から必要なファイルをマウスで中クリック (中央のボタンを押す) してください。

ファイル名を入力する時には、最後までファイル名を入力しなくても、先頭の 1〜2 文字を入力して、スペースキーを押すだけで、 残りの部分を自動的に入力 (補完) してくれます。

編集中のファイルを上書き保存するには、“C-x C-s” と押します。また、別名で保存するには、“C-x C-w” と押した後に、

```
Write file:  ~/
```

と表示されるので、ファイル名を入力すれば OK です。

3.7 切り取り/貼り付け

カーソルのある行を切り取るには、“C-k” と押します。切り取った行の内容を別の場所に貼り付けるには、貼り付ける場所にカーソルを移動して、“C-y” と押します。

行をコピーしたい場合には、“C-k C-y” で、その場で切り取り・貼り付けを行った後に、コピーしたい場所にカーソルを移動して “C-y” と押せば OK です。

“C-k” を複数回押すことで、複数の行を切り取ることができます。このようにして切り取った内容は、“C-y” を押すことで一度にまとめて貼り付けることができます。

また、マウスの左ボタンでドラッグして範囲を指定した後、メニューの [Edit] → [Cut] で切り取り、[Edit] → [Copy] でコピーなどを行うこともできます。移動またはコピーしたい場所で [Edit] → [Paste] すれば貼り付けることもできます。

3.8 Emacs 入門ガイドを使ってみよう!

Emacs の使い方は、“C-h T” (“C-h” を押 してから、コントロールキーを離して “T” を押す) と押すと見ることができます。

3.9 わからなくなったら？

ウィンドウの一番下の行に何か表示されて (例えば「C-h (Type ? for further options)-」など)、何をしたら良いのかわからなくなる事があります。その時は、“C-g” を押すことで (何回か押す必要があるかもしれません) 普通の状態に戻ることができる場合があります。それでもうまく行かなかったら、担当教官に相談してください。

第 4 章

C プログラムの作成・コンパイル・実行

4.1 プログラム作成の流れ

C 言語で記述されたプログラムを作成・実行するためには、まずエディタでプログラムの内容を記述し、ファイルとして保存しなければなりません。この時、ファイル名には「.c」という拡張子を付けます。例えば、「hello.c」 というような名前で保存します。このようなファイルを**ソースファイル**と呼びます。しかし、このソースファイルと呼ばれるものは単なるテキストファイル (人間が簡単に中身を理解するための最も一般的な形式) であるため、プログラムを実行することはできません。

ソースファイルをコンピュータが理解できる**実行形式** (マシン語によるプログラム) へと翻訳 (**コンパイル**) する必要があります。コンパイルするためには、**コンパイラ**と呼ばれるプログラムを実行する必要があり、ここでは「gcc」というコンパイラを利用します。ソースファイルをコンパイルすると実行形式のファイル (**実行ファイル**) が作成され、実行ファイルのファイル名を入力すればプログラムが実行できます。

4.2 エディタでソースファイルを作成

まず、作業用のディレクトリを作ります。課題の場合は、課題番号 (T01 など) をディレクトリ名にします。「cd T01」 でカレントディレクトリを変更した後、C 言語のソースプログラムをエディタで作成します。

例として、C 言語の最も基本的なプログラムを入力してみましょう。

プログラム 4.2.1 hello.c

```
/* 最も基本的な C のプログラム */
#include <stdio.h>
int main()
{
  printf("Hello, world !\n");
  return 0;
}
```

これは画面に “Hello world !” と表示するプログラムです。まず、Emacs (エディタ) を用いてプログラムを入力し、ソースファイルを作成します。ソースファイル名は「hello.c」とします。

```
b00b0yy@txx:~/T01$ emacs hello.c &
```

つぎにプログラムを入力します。入力が終わりましたら保存します。端末で ls を実行して「hello.c」が新しく作成されているのを確認してください。

```
b00b0yy@txx:~/T01$ ls
hello.c
```

4.3 コンパイル

コンパイルするには端末で

```
gcc ソースファイル名
```

と入力します。先程作成した「hello.c」をコンパイルする場合は、

```
b00b0yy@txx:~/T01$ gcc hello.c
```

のようにします。ソースファイルの記述に文法上の誤りがなければ、何もメッセージが表示されずに、「a.out」という実行ファイルが生成されます。誤りがある場合は、行番号を含むメッセージがたいてい表示されるのでソースファイルを修正、保存した後、再度 gcc でコンパイルを行います。これをエラーメッセージが表示されなくなるまで繰り返します。この繰り返しをデバッグといいます。エラーがなくなったら、もう一度 ls を実行します。

```
b00b0yy@txx:~/T01$ ls
a.out*    hello.c
```

「a.out」というファイルが作成されています。ここまできて、初めて作成したプログラムを実行することができます。なお、ls で表示したときにファイル名の後ろの 「*」 は、そのファイルが実行ファイルであることを意味します。

実行ファイルの名前を「a.out」ではなく任意に指定するには gcc -o **実行ファイル名 ソースファイル名**のようにします。

実行ファイル名は一般には、ソースファイルの拡張子を取ったものを指定します。ソースファイルが「hello.c」の場合は、実行ファイル名は「hello」とします。

```
b00b0yy@txx:~/T01$ gcc -o hello hello.c
```

こうすると、どのソースファイルの実行ファイルがどれかが一目でわかります。ls で確認してみると

```
b00b0yy@txx:~/T01$ ls
a.out*    hello*    hello.c
```

のように「hello」という実行ファイルが作成されています。

4.4 プログラムの実行

作成したプログラムは、

./実行ファイル名

というコマンドで実行できます。たとえば、実行ファイル「hello」を実行したいときは

```
b00b0yy@txx:~/T01$ ./hello
Hello, world !
```

のようにします。

なお、画面に実行結果を表示させずにファイルに出力するように実行するには、

./実行ファイル名 > 実行結果を入れるファイル名

のようにします。

```
b00b0yy@txx:~/T01$ ./hello > hello.output
```

この場合は「hello.output」というファイルに実行結果が文字として書き込まれます。lv を用いて「hello.output」の中身を見ると

```
b00b0yy@txx:~/T01$ lv  hello.output
Hello, world !
```

となり、実行結果が「hello.output」ファイルの中に書き込まれているのがわかります。

また、プログラムがキー入力によるデータを必要とするときに、キー入力の代わりにファイルに入力データを書いておけば

./実行ファイル名 < 入力内容が入ったファイル名

とすることで、プログラムにデータを入力することができます。

4.5 デバッグ

ソースプログラムを入力して、コンパイルすればプログラムは完成しますが、一回でこれらの作業が完了することはまれです。普通は、バグと呼ばれるプログラム上のエラーが、コンパイルや実行の段階で発生します。このバグを見つけて正しくソースプログラムを修正することをデバッグといい、プログラミングでは非常に重要な位置を占めます。プログラミングはこのデバッグにほとんどの時間を費やすと言っても過言ではありません。

少し複雑なプログラムを作成してコンパイルすると、何十行にもわたってエラーメッセージが表示されることがあり

ます。初心者はこれを見てあわてふためきますが、心配することはありません。エラーにはたいていソースファイルの行番号が表示されるので、どの行でエラーが発生したかを知ることができます。ここで、多くの初心者はエラーメッセージの一番最後のエラーをみてその行を修正しようとしますが、ほとんどの場合その行にはエラーはありません。例えば、プログラムの最初の方で行末のセミコロン「;」を忘れてしまったとします。コンパイラはその行と次の行が続いていると見なしますので、本来の命令とは異なって翻訳しエラーとなります。このエラーがその次の行の翻訳にもまたその次の行にも影響し... というように 1 つのエラーが他のエラーを誘発することがしばしばあり、何十行にもわたるエラーメッセージになります。このことからわかるのは、デバッグはエラーメッセージの一番最初のエラーから行うのがよいということです。このように、初心者はデバッグに多少苦労しますが、何度か経験すれば対処の仕方が分かってきます。

また、コンパイル時にはエラーが発生しなくても、実行結果が正しくない場合は、アルゴリズム (考え方) の検討にまでさかのぼることもあり、非常にやっかいなことになります。さらに、自然科学系のプログラミングの場合、実行結果が正しいかどうかを吟味する力が必要になります。すなわち、結果が物理的に正しいかを判別するには物理学や数学などの基礎学力がものをいいます。プログラマーになるから物理や数学を勉強する必要はない、という考えは全く見当違いです。

4.6 コンパイルと make

実行ファイルの名前を「a.out」ではなく任意に指定して、ソースファイルをコンパイルするには

```
gcc   -o   実行ファイル名   ソースファイル名
```

としましたが、これには少し問題があります。コンパイルの時、間違って

```
gcc   -o   ソースファイル名   ソースファイル名
```

としてしまうと、せっかく作ったソースファイルが実行ファイルで上書きされ、消えてしまうのです。たとえば、先の例のように「hello.c」というソースファイルから、「hello」という実行ファイルを作る場合を考えましょう (**下記の例は実行しないでください**)。

```
b00b0yy@txx:~/T01$ ls
a.out*    hello*    hello.c
b00b0yy@txx:~/T01$ gcc  -o  hello.c  hello.c
b00b0yy@txx:~/T01$ ls
a.out*    hello*    hello.c*
b00b0yy@txx:~/T01$ lv  hello.c
"hello.c" may be a binary file.  See it anyway?
```

となってしまい、hello.c は読めなくなってしまいました。このような問題は、「make」というコマンドを使って防ぐことができます。make は簡単に言うと、「人間にかわってコンパイラなどのコマンドを実行してくれるコマンド」で、プログラムのコンパイルには多くの場合 make が使用されます。最も簡単に make を使うには

```
make 実行ファイル名
```

とします。実行ファイル名には、ソースファイルから拡張子 「.c」 を取ったものを指定します。例えば、「hello.c」 をコンパイルするには、以下のようにします。(以下の例を実行する前に、先ほど作った実行ファイル「a.out」,「hello」を消去 (rm) しておいて下さい)

```
b00b0yy@txx:~/T01$ make hello
   cc hello.c -o hello
```

これによって、「hello.c」 がコンパイルされ、実行ファイル「hello」が作成されます。また、間違って make の引数にソースファイル名「hello.c」 を指定してしまっても

```
b00b0yy@txx:~/T01$ make hello.c
make: 'hello.c' に対して行うべき事はありません.
```

と表示され、 「hello.c」 という実行ファイルが作成されることはありません。また、すでに実行ファイルができており、実行ファイルがソースファイルよりも新しければ

```
b00b0yy@txx:~/T01$ make hello.c
make: 'hello' は更新済みです.
```

のように表示され、無駄なコンパイルは行われません。

4.7 Makefile

make は一般的には 「Makefile」 という名前 (**先頭の "M" は大文字です**) のファイル (**メイクファイル**と呼ばれます) と共に使用します。「hello.c」のコンパイルに利用するメイクファイルはつぎのような内容のテキストファイルです。

プログラム 4.7.1 Makefile

```
CC = gcc
all : hello
```

「Makefile」というファイルを上記のように作成し、プログラムのソースファイルと同じディレクトリに保存した後、引数なしで make コマンドを実行すると、自動的にコンパイラを適切なコマンドで実行してくれます。

```
b00b0yy@txx:~/T01$ ls
Makefile    hello.c
b00b0yy@txx:~/T01$ make
gcc hello.c -o hello
```

メイクファイルを用いるとソースファイル名や実行ファイル名を指定する必要がないので、入力を間違うことはありません。また、デバッグで何回も同じソースファイルをコンパイルするときも、単に make と打つだけで済みます。

一般的にメイクファイルはつぎのように書きます。

設定行（マクロ行と言うこともあります）
設定行
　:
　:

ルール行（依存関係行と言うこともあります）
[Tab] **コマンド行**

ルール行
[Tab] **コマンド行**
　:
　:

先の例では、CC = gcc が設定行で、all :　hello というのがルール行です。コマンド行はありません。
設定行は

設定項目 = 設定する内容

という形式で書きます。設定行は以下のようにして使用します。

プログラム 4.7.2 Makefile (設定行)
```
CC = gcc
LDFLAGS = -lm
all : hello
```

この例では、「C 言語のコンパイラとして gcc を使う」という設定と、「コンパイルした結果のマシン語のプログラムと、m ライブラリ (システムにあらかじめ用意されている数学関数を処理するためのプログラム、マシン語で記述されている) を結合して実行形式を作成する」という設定を行っています。

コンパイラ (gcc) の内部では、リンカ (ld) というプログラムが動作しています。リンカは複数のマシン語のプログラムを結合して一つの実行形式を作成するためのプログラムです。設定項目 LDFLAGS は、このリンカに与える設定項目を示していて、“-l (なんとか)” と書くと、「(なんとか) ライブラリを結合して実行形式を作成する」という設定を表します。

ルール行は、ちょうど料理のレシピのような内容で

ターゲット :　ターゲットを作るのに必要なもの（材料）

という形式で書きます。材料が複数必要な場合は、スペースで区切って並べます。all :　hello と書くと、「all (全部完成という意味のターゲット) を作るには、hello (という名前の実行ファイル) ができていなくてはいけない」ということを指示することになります。例えば、「hello」 という実行ファイルを作るのに 「hello.c」 というソースファイルが材料として必要であるということを示すには、all :　hello という行の下に hello :　hello.c というように書けばよいのですが、このようにあきらかなルールは省略できます。

コマンド行は、直前のルール行で示されたターゲットを作るためのコマンド (Unix の命令) を指示するための行です。必ず先頭に [Tab] (タブキーを押したときに挿入される空白) を挿入し、その後にコンパイルの時に用いるコマンドを書きます。

コマンド行は以下のように使用します。

プログラム 4.7.3 Makefile (コマンド行)

```
CC = gcc
LDFLAGS = -lm
all : hello
        $(CC) $(LDFLAGS) hello.c -o hello
```

この例では、「hello.c」 というソースファイルから 「hello」 という実行形式を作成するために gcc コマンドをどのように使用するかが示されています。メイクファイルの中では、“$(設定項目)” という文字列は、設定行で設定された内容に置き換えられます。したがって上記の例のコマンド行は

```
gcc  -lm  hello.c  -o  hello
```

と書いてあったのと同じということになります。そこで、「hello.c」 から「hello」を作るには gcc を -l という引数付きで実行すれば良いということがわかります。

実際には、ルール行が決まればコマンド行はそれに応じてほぼ自動的に決まるので、ほとんどの場合にはコマンド行は省略できます。つまりプログラム 4.7.2 のように記述すれば十分です。

同じディレクトリの中で複数の実行ファイルを作りたい時は、all : の後に作りたい実行ファイル名をスペースで区切って並べます。

プログラム 4.7.4 Makefile (複数の実行ファイルの作成)

```
CC = gcc
LDFLAGS = -lm
all : hello bye
```

メイクファイル (「Makefile」) は、必ずプログラムのソースファイルと同じディレクトリに置き、そのディレクトリをカレントディレクトリにして make コマンドを実行します。このように メイクファイルを作っておくと、make と打つだけでそのディレクトリ内で作らなくてはいけない実行ファイルを一度に全部作ってくれます。

```
b00b0yy@txx:~/T01$ ls
Makefile    bye.c    hello.c
b00b0yy@txx:~/T01$ make
gcc -lm hello.c -o hello
gcc -lm bye.c -o bye
b00b0yy@txx:~/T01$ ls
Makefile    bye*    bye.c    hello*   hello.c
b00b0yy@txx:~/T01$ ./hello
Hello, world !
b00b0yy@txx:~/T01$ ./bye
Good-bye, world !
```

演習ではメイクファイルを作成してからコンパイルするように心がけましょう。

第 5 章

課題の提出方法

5.1 課題を提出しよう

課題に対する解答を作成したら、早速提出してみましょう。提出は Unix マシン上で電子的に行います。通常、提出しなくてはならないのは、プログラムの**ソースファイル** (「〜.c」 のファイル) です。間違えて実行形式のファイルを提出しても採点の対象にはなりませんので注意してください。セメスタ課題などでは、それに加えて考察を書いたファイルの提出が必要になります。課題を提出するには、「**課題番号**」と「**問題番号**」が必要になります。「課題番号」と「問題番号」は、問題用紙に書いてありますのでチェックしてください。

提出手順は、まず、作成したプログラムのソースファイルのあるディレクトリがカレントディレクトリになってなくてはいけません。ls コマンドで、カレントディレクトリに提出するソースファイルがあるか、確認してください。

```
b00b0yy@txx:~/T01$ ls
hello.c    hello*
```

ここでは、hello.c が提出すべきソースファイルであるとします。また、「課題番号」は T01、「問題番号」は 2 であったとします。提出は submit というコマンドで行います。

```
b00b0yy@txx:~/T01$ submit  T01  2  hello.c
課題番号 :  T01
問題番号 :  2
hello.c を提出します。
hello.c を提出しました。(Tue Apr 1 16:18:20 JST 2008)
```

上記のように表示されたら、提出は完了です。submit コマンドは

```
submit   課題番号   問題番号   提出ファイル名
```

と入力します。

提出は、締切前であれば**何度でも**行うことができます。ですから、一度提出した後にもう一度見直して間違いを見つけた場合などは、修正したものを提出しなおしてください。

```
b00b0yy@txx:~/T01$ submit T01 2 hello.c
課題番号 :  T01
問題番号 :  2
hello.c を提出します。
hello.c と同じ名前のファイルが既に提出されています。
hello.c を提出すると、以前に提出したものは消去されます。
このファイルを提出しますか？ [y/n]:  y
hello.c を提出しました。(Tue Apr 1 16:27:47 JST 2008)
```

提出しなおした場合には、以前に提出したものは消去され新しく提出したものだけが残ります。念のため本当に提出するか確認しますので、新しく提出するのであれば 「y」 を、そうでなければ「n」を入力してください。

5.2 締め切り

提出の締め切りを過ぎた場合には、つぎのように表示されます。

```
b00b0yy@txx:~/T01$ submit T01 2 hello.c
課題番号 :  T01
問題番号 :  2
この問題の提出は締め切りました。
提出の遅延を希望する場合は、担当の教官と相談してください。
```

この場合には提出は受け付けられません。提出の遅延を希望する場合は、担当の教官と相談してください。ただし、**基本課題については提出の遅延は認められません。**必ず締め切り前に提出するように、特に注意してください。

5.3 提出の確認

提出されたファイルは、提出箱ディレクトリという特別なディレクトリに入れられます。自分が提出したファイルが正常に提出箱に入っているかどうかは、check コマンドで確認できます。check コマンドは

```
check　課題番号　問題番号　　確認するファイル名
```

と入力します。確認したいファイルの内容が長い場合は、つぎのように | (縦棒:「パイプ」と呼ばれます) と lv コマンドを併用し、一画面づつ区切って表示するようにすると良いでしょう。

```
b00b0yy@txx:~/T01$ check T01 2 hello.c | lv
課題番号 : T01
問題番号 : 2
この問題の提出は締め切りました。
提出の遅延を希望する場合は、担当の教官と相談してください。
堤出されたファイルの履歴を確認します。
-------------------------------------------------
Tue Apr  1 16:18:20 JST 2008  hello.c  b00b0yy
-------------------------------------------------
==============================
  hello.c の内容が表示される
==============================
```

5.4 評価の確認と再提出

課題の提出締め切り後、一週間以内であれば各自の判断で再提出 (任意) を行うことができます。判断の材料として、プログラミング演習の Web ページ
http://www.ec.h.akita-pu.ac.jp/~programming/
から確認できる採点結果を活用してください。

また、再提出の締め切り前に、担当の教官から再提出を行う際に留意すべきヒントが与えられることがあるので、適宜活用してください。再提出のためのヒントは、主に「README」というファイルによって与えられます。ただし、その他のファイルに記述が及ぶ場合もありますので、Web ページに提示された内容をよく読んでください。

「README」 に書かれたヒントを読むには、以下のようにします。

```
b00b0yy@txx:~/T01$ check T01 2S README | lv
課題番号 : T01
問題番号 : 2S
この問題の提出は締め切りました。
提出の遅延を希望する場合は、担当の教官と相談してください。
堤出されたファイルの履歴を確認します。
-------------------------------------------------
Tue Apr  1 16:18:20 JST 2008  hello.c  b00b0yy
-------------------------------------------------
==============================
  README の内容が表示される
==============================
```

また、提出前や再提出前にヒントが欲しい場合は、担当教官に直接質問するか、メールで質問してください。メールアドレスはプログラミング演習の Web ページに記載されています。

ヒントに従って堤出するファイルを修正し、再提出する場合は、「課題番号」は問題用紙と同じ番号ですが、「問題番号」は問題用紙に書いてある番号の後に R を付けます。

```
b00b0yy@txx:~/T01$ submit T01 2R hello.c
課題番号 :  T01
問題番号 :  2R
hello.c を提出します。
hello.c を提出しました。(Tue Apr 1 16:18:20 JST 2008)
```

第 II 部

プログラミング演習 スタイル規則

第 6 章

スタイル規則とは

プログラミングにおいては、要求仕様を満すだけではなく、可読性・保守性に優れたコードを記述することが重要です。現在のソフトウェア開発においては、小規模なプログラムを 1 人が短期間で作成する能力だけでなく、大規模なソフトウェアを多人数が長期間かけて作成する能力が求められます。多人数でソフトウェアを作成する場合、たった 1 人でも可読性・保守性の悪いコードを記述すれば、ソフトウェア全体の作成期間が延るだけでなく品質も悪化します。また、1 人でプログラムを作成する場合でも、可読性・保守性に優れたコードを記述するように心がけていれば、プログラムにバグが入り込む危険性は、意識しない場合より格段に少くなります。このように、可読性・保守性の高いコードを記述する能力は、プログラミング能力のなかで重要な要素の一つです。

ただ、プログラミング初心者にとっては、可読性・保守性の高いコードの記述は容易ではありません。しかし、先人プログラマ達によって、良質なコードを記述しやすい経験的規則が見つけられています。それらの規則をまとめたものが、スタイルです。ある一定のスタイルに従ってコードを記述すれば、初心者であっても可読性・保守性の高いコードを記述できます。

本演習では、本演習独自のスタイル規則を定めます。

スタイルにはプログラムの規模や用途、開発の形態に応じていくつもの流派があり、全世界共通普遍のスタイルはありません。ここで定めるスタイル規則は世間一般で良いとされるスタイル規則のひとつに基づいて作成していますが、あくまでこの演習でのスタイルであることに注意して下さい。この演習で定めるスタイル規則は、実社会でのソフトウェア開発の現場で利用されているスタイル規則と比べると、教育上重要と考えられる点だけに着目した、単純な規則となっています。初心者の間違いを少くするためのスタイル規則もあります。また、全体としてプログラムの説明をできるだけ多く記述するような規則としてあります。

一部のスタイル規則は、プログラミングに慣れた人には面倒に感じるかもしれません。しかし、プログラミングの現場で与えられたスタイルに対応した可読性の高いプログラムを記述する能力を身につけるため、この演習では提出されたプログラムが以下に掲載するスタイル規則に従っているかどうかを評価の基準のひとつとします。

第 7 章

スタイル規則分類

ここでは、スタイル規則を大まかに分類して示します。普段は以下の項目だけを意識してプログラミングを行なえばよいでしょう。演習の最初の方ではわからない項目もあるでしょうが、演習が進むにつれて必要なスタイル規則が明かになるでしょう。スタイル規則は意識して欲しい順に並べたつもりです。ただし、**F.** 細則は、分類に入らないような全体的な規則を集めたものですので、いつも注意するようにして下さい。

A. **コメント**　プログラムに適切な説明を加えるための規則です。
B. **命名**　プログラム中のさまざまな要素に対し，適切な命名を行うための規則です。
C. **インデント**　プログラムの構造をわかりやすくするための字下げを行なうための規則です。
D. **型**　適切な型を用いるための規則です。
E. **演算子**　演算子の範囲と効果を適切に制御するための規則です。
F. **細則**　上記以外の規則です。

以降では、各分類にしたがって、本演習でのスタイル規則とその説明を書きます。個々の規則は、枠線で囲って示します。なお、これらの規則は、演習時間内に行なう練習用のプログラムでは、厳密に守らなくてもかまいません。ただし、提出された課題は、コードの可読性・保守性の高さ、すなわちスタイルによっても評価が変わることに注意して下さい。(もちろん、要求仕様の実現度によっても評価は変ります。)

このスタイル規則中ではプログラムのうち説明に必要な部分のみを抜き出して示しています。「良い例」として示されているプログラムも、完全な動作するプログラムとはなっていない場合がありますので注意してください。特に、「A. コメント」より後の節で例として示しているプログラムでは、コメントは説明に必要なものしか示していません。「A. コメント」の節と相互に参照し、足りない部分を補って考えてください。

A　コメント

コメント（プログラムのソースファイルの各所に記述する説明）は、プログラムの可読性を高めるのに有用です。必要最低限のコメントを入れることは難しいのですが、初心者はコメント不足のプログラムよりもコメントの十分入った記述を心がけましょう。近くの人達でプログラムを見せあって、自分のプログラムが他人に理解できるか確かめるといいでしょう。

A-1　プログラム自身の説明

プログラム自身の説明をソースファイルの先頭にコメントとして挿入すること。その際、学籍番号、名前は必ず書くこと。(採点時に利用しますので、学籍番号、ソースファイル名などは必ず 1 バイト英数小文字で記述してください。)

プログラム A.1 良い例

```
/*
 * 作成日 : 2008/4/1
 * 作成者 : 本荘太郎
 * 学籍番号 : b00b0yy
 * ソースファイル : gcd.c
 * 実行ファイル : gcd
 *
 * 説明 : ユークリッドの互除法により最大公約数を求めるプログラム
 *
 * 入力 : 二つの正整数値 a, b を標準入力から受け取る。
 *        a と b はどちらが大きくとも良い。
 * 出力 : a と b の最大公約数をユークリッドの互除法により
 *        求めた結果を標準出力に表示する。
 *        最大公約数は正整数である。
 *        a または b が 0 以下である場合には、最大公約数は
 *        存在しないため、エラーメッセージを表示して終了する。
 */

int main()
{
```

プログラム A.2 悪い例

```
int main()
{
```

A-2 変数の説明

新しい変数を宣言するときには、その意味をコメントとして挿入すること。

プログラム A.3 良い例

```
int main()
{
  int   season; /* 季節を表す整数(0:春,1:夏,2:秋,3:冬) */
  double kion;  /* 気温(摂氏) */
```

プログラム A.4 悪い例

```
int main()
{
  int   season;
  double kion;
```

A-3 関数の説明

(main 関数以外の関数では、)
関数の機能説明、仮引数の説明、および戻り値の説明をコメントとして挿入すること。プロトタイプ宣言の箇所にも適切なコメントを記述すること。

プログラム A.5 良い例

```
/* プロトタイプ宣言 */
int remainder(int x, int y); /* 被除数 x を除数 y で割った余りを求める関数 */



/*
  被除数 x を除数 y で割った余りを求める関数
  引数
    x : 被除数（整数）
    y : 除数（0 以外の整数）
  戻り値 : x を y で割った余り（ 商*除数 + 余り = 被除数 となる値）である整数を返す．
           ただし， |商*除数| <= |被除数| であるものとする．
*/
int remainder(int x, int y)
{
```

プログラム A.6 悪い例

```
int remainder(int x, int y);

int remainder(int x, int y)
{
```

B 命名

変数や関数の命名は、プログラムの可読性に重要な役割を果たします。適切は命名は、プログラムの理解を助け、デバッグ作業を潤滑にしてくれます。一方、意味のない記号で命名したり、変数や関数の意味が命名された文字列の意味と異ったりすると、可読性が著しく低下します。

B-1 命名の一般的規則

変数や関数は、そのデータや機能の意味が類推できるように命名すること。

プログラム B.1 良い例

```
int main()
{
  int    season; /* 季節を表す整数(0:春,1:夏,2:秋,3:冬) */
  double kion;   /* 気温(摂氏) */
```

プログラム B.2 悪い例

```
int main()
{
  int    a;       /* 季節を表す整数(0:春,1:夏,2:秋,3:冬) */
  double data;    /* 気温(摂氏)  */
```

変数にはいくつかの種類がありますが、それらを一目で区別するために、いくつかのスタイル規則を定めます。

B-2 ローカル変数名

ローカル変数 (main 関数内の変数も含む) は、英小文字、数字、アンダースコア (_) だけで命名すること。

プログラム B.3 良い例

```
int main()
{
  int    season; /* 季節を表す整数(0:春,1:夏,2:秋,3:冬) */
  double kion;   /* 気温(摂氏) */
```

プログラム B.4 悪い例

悪い例

```
int main()
{
  int   Season; /* 季節を表す整数（0:春,1:夏,2:秋,3:冬） */
  double K;     /* 気温（摂氏） */
```

B-3　マクロ名

マクロは、英**大文字**、数字、アンダースコア (_) だけで命名すること。

プログラム B.5 良い例

良い例

```
#define  HARU   0
#define  NATU   1
#define  AKI    2
#define  FUYU   3
```

プログラム B.6 悪い例

```
#define  haru   0
#define  Natu   1
#define  akI    2
#define  FuYu   3
```

B-4　関数名

関数名は、英**小文字**、数字、アンダースコア (_) だけで命名すること。

プログラム B.7 良い例

```
int remainder(int x, int y);

int remainder(int x, int y)
{
```

プログラム B.8 悪い例

```
int Remainder(int x, int y);


int Remainder(int x, int y)
{
```

B-5 グローバル変数名

グローバル変数名は、最初の文字だけを英**大文字**とし、残りは英**小文字**、数字、アンダースコア (_) にすること。

プログラム B.9 良い例

```
double Heikin; /* 平均 */
double Bunsan; /* 分散 */
int main()
{
```

プログラム B.10 悪い例

```
double heikin; /* 平均 */
double BUNSAN; /* 分散 */
int main()
{
```

B-6 新しい型名

typedef によって新しい型を定義するときには、最初の文字だけを英大文字とし、残りは英小文字と数字とすること。

プログラム B.11 良い例

```
typedef   int Week; /*曜日を整数で表す型*/
int main()
{
  Week  today;    /*今日の曜日を扱う変数*/
```

プログラム B.12 悪い例

```
typedef   int week; /*曜日を整数で表す型*/
int main()
{
  week  today;   /*今日の曜日を扱う変数*/
```

C インデント

インデントは、プログラムの構造をそのコード内で視覚的に表現する一つの方法です。適切なインデントを行なえば、可読性および保守性ともに向上します。C 言語では、ある一連の処理を中括弧 ({ と }) で囲んで記述します。これを複文といいます。
C 言語におけるインデントは中括弧とその中身に関する字下げ等の規則です。インデントにもいろいろ流儀がありますが、下の規則に従えば、本演習でのインデントが得られます。なお、エディタとして Emacs を用いている場合には、最初の行から順に [Tab] キーを押していけば、きれいなインデントが得られます。

C-1 中括弧の中身の記述

中括弧の内部は、対応する中括弧よりも 1 段字下げすること。中括弧の内部で中括弧が入れ子になっていないもの (同じレベルのもの) は、左端をそろえて記述すること。

C-2 制御構造における複文の利用

選択や繰り返しなどの制御構造において、その中身 (被制御部分) は複文とすること。つまり、if 文、while 文、for 文などでは、条件判断の後に必ず中括弧を入れること。

プログラム C.1 良い例

```
int main()
{
  int i;    /* カウンタ*/

  for(i=1;i<10;i++)
  {
    if(i%2==0)
    {
      printf("i=%d\n",i);
    }
  }
  return 0;
}
```

プログラム C.2 悪い例

```
int main(){
  int i;    /* カウンタ*/

  for(i=0;i<10;i++){
    if(i%2==0)printf("i=%d\n",i);}
  return 0;}
```

D 型

適切な型を用いることは、コードの保守性に重要な役割を果します。特に、整数と実数の型に違いに注意して下さい。数学では整数と実数の区別をあいまいに扱っても正しい答えが得られることも多いですが、プログラミングでは整数と実数の区別をしっかりしないととんでもない結果になることがあります。
実際、どんなプログラミング言語でも、理想的な実数は扱うことができず、実数を近似した浮動小数点数だけを扱うことができます。C 言語では、整数と浮動小数点数のデータ表現形式はまるで違うので、それらは明確に区別しなければなりません。

D-1 型の利用

適切な型を用いること。

例に示すように、連続的な量を扱う場合は double 型を用いて、離散的な量を扱う場合は int 型を用いて下さい。

プログラム D.1 良い例

```
int main()
{
  /*変数宣言*/
  double suion;   /*水温（摂氏）*/
  int today_week; /*今日の曜日*/
  /*(0:日,1:月,2:火,3:水,4:木,5:金,6:土)*/
```

プログラム D.2 悪い例

```
int main()
{
  /*変数宣言*/
  int suion;   /*水温（摂氏）*/
  double today_week; /*今日の曜日*/
  /*(0:日,1:月,2:火,3:水,4:木,5:金,6:土)*/
```

実は、C 言語には、整数を扱う型が (int 型だけでなく) 複数あり、実数 (浮動小数点数) を扱う型が (double 型だけでなく) 複数あります。しかし、初心者にとって、これらの型の違いは繁雑なだけです。ですから、本演習では、代表的な型だけを用いることにします。

D-2 整数

整数を扱う変数では、int 型だけを用いること。

D-3　実数

実数を扱う変数では、double 型だけを用いること。

D-4　論理値

論理値を扱う変数は、int 型だけを用いること。

プログラムは、条件が正しいか間違っているかによって制御されます。いいかえると、プログラムは、真、偽の 2 値論理に基づいて動きます。したがって、プログラミング言語によっては、論理値だけを扱う論理型が特別に準備されているものもあります。しかし、C 言語では、論理値だけを扱う型がなく、整数型で代用します。つまり、整数値として、0 を持つとき偽を表し、0 以外を持つとき真を表します。

プログラム D.3 良い例

```
/*変数宣言*/
double suion; /*水温（摂氏）*/

if(suion!=0.0)
  {
    printf("水温は 0 度ではありません。\n");
  }
```

プログラム D.4 悪い例

```
/*変数宣言*/
double suion; /*水温（摂氏）*/

if(suion)
  {
    printf("水温は 0 度ではありません。\n");
  }
```

D-5　定数

適切な定数型を用いること。

結構忘れがちなのが定数にも型があることです。数学の場合には、“3” と書けば “整数の 3” かもしれないし、“実数の 3” を意味しているかもしれません。しかし、C 言語では、“3” は “整数の 3” を表します。C 言語では、“実数の 3” を

表すには、“3.0” のように書かなければいけません。特に、実数を表す定数には、小数点を付けようにしましょう。

プログラム D.5 良い例

```
printf("水温は %lf 度です。\n",0.0);
```

プログラム D.6 悪い例

```
printf("水温は %lf 度です。\n",0);
```

E　演算子

C 言語のプログラムでは、演算子が中心的な役割を果たします。C 言語においての演算は、算術計算だけを意味しているのではありません。例えば、変数へ値を代入することも演算に他ならず、代入演算子「=」が用いられます。正しい C 言語のプログラムは、演算子を適切に制御して得られます。したがって、保守性の高いコードを記述するためには、演算子の効果を深く理解することがかかせません。しかし、演算子の効果を深く理解することは簡単ではありません。そこで、初心者にとっても、プログラムにバグが入り込みにくいように、演算子の利用規則を定めます。特に、演算子によっては型を (コンパイラが) 自動で変換してしまったりしますので、それらの結果を (プログラミングした本人でさえ) 把握しにくくなったりします。

E-1　型変換 (キャスト演算子)

型変換はキャスト演算子を用いること。(自動型変換は用いないこと。)

規則 **E-1** を具体的に述べると規則 **E-2** のようになります。

E-2　2 項演算子

2 項演算子の両辺 (2 つの被演算項) は同じ型であること。

次の規則 **E-3, E-4** は、規則 **E-2** に含まれるのですが、特に注意して欲しい事項ですので、別途規則として定めます。

E-3　代入演算子

代入演算子「=」の両辺は同じ型であること。

E-4　算術演算子

「*」,「+」,「-」,「/」等の算術演算子の 2 つの被演算項は同じ型であること。

プログラム E.1 良い例

```
/*変数宣言*/
int kou;          /*第何項目かを表わす変数*/
double gyakusu;   /*kou の逆数を表す変数*/

gyakusu=0.0;
gyakusu=1.0/( (double) kou );
```

プログラム E.2 悪い例

```
/*変数宣言*/
int kou;          /*第何項目かを表わす変数*/
double gyakusu;   /*kou の逆数を表す変数*/

gyakusu=0;
/* 代入演算子「=」の左辺が double 型、
   右辺が int 型になってます。*/
gyakusu=1.0/kou;
/* 算術演算子「/」の左の被演算子が double 型、
   右辺が int 型になってます。*/
```

F　細則

ここでは、A〜E までの規則に分類しなかったけれども、必ず守ってもらいたい規則を示します。

F-1　main 関数の型

(教科書では main の型は void 型ですが、)
main 関数の型は int 型にすること。

F-2　関数と return 文

関数は必ず return 文で終了すること。

特に、main 関数の最後に「return 0;」の記述を忘れないこと。また、関数の戻り値が void 型のときでも、「return;」の記述は忘れないこと。

プログラム F.1 良い例

```
void print(int x); /* 整数型の変数の内容を表示する関数 */

int main()
{
  int i;
  print(i);
  return 0;
}

void print(int x)
{
  printf("%d\n");
  return;
}
```

プログラム F.2 悪い例

```
void print(int x); /* 整数型の変数の内容を表示する関数 */

main()
{
  int i;
  print(i);
}

void print(int x)
{
  printf("%d\n");
}
```

第 III 部

付録

付録 A

Unix の基本的なコマンド

A　ディレクトリ操作関連

カレントディレクトリを表示	`pwd`
ディレクトリの作成	`mkdir` **ディレクトリ名**
カレントディレクトリの内容を表示	`ls`
指定したディレクトリの内容を表示	`ls` **ディレクトリ名**
カレントディレクトリの変更	`cd` **ディレクトリ名**
カレントディレクトリを一つ上のディレクトリに変更	`cd ..`
カレントディレクトリをホームディレクトリに変更	`cd`
ディレクトリの消去[*1]	`rmdir` **ディレクトリ名**

B　ファイル操作関連

ファイル内容の表示	`lv` **ファイル名**
ファイルを別名のファイルとしてコピー	`cp` **コピー元ファイル名　コピー先ファイル名**
ファイルを別のディレクトリにコピー	`cp` **コピー元ファイル名　コピー先ディレクトリ名**
ファイル名を変更	`mv` **現在のファイル名　新しいファイル名**
ファイルを別のディレクトリに移動	`mv` **移動すべきファイル名　移動先ディレクトリ名**
ファイルを消去	`rm` **ファイル名**
ファイルの編集	`emacs` **ファイル名** `&`

[*1] **rmdir** で消去できるのは中にファイルやディレクトリが一つもないディレクトリのみです

C プログラム開発

C 言語で書かれたプログラムのコンパイル (実行ファイル名 : a.out)	`gcc` **ソースファイル名**
C 言語で書かれたプログラムのコンパイル (実行ファイル名を指定)	`gcc -o` **実行ファイル名 ソースファイル名**
make によるコンパイル (実行ファイル名を指定)	`make` **実行ファイル名**
メイクファイルを用いたコンパイル	`make`
カレントディレクトリにある実行ファイルの実行	**./実行ファイル名**

D その他

パスワードの変更	`yppasswd`
現在の時刻を表示	`date`
コマンドのマニュアルを参照	`man` **コマンド名**

E シェルの機能

ヒストリ (過去に入力したコマンドの表示)	カーソルキー ([↑],[↓]),“C-p”,“C-n”
行内編集機能	カーソルキー ([←],[→]),“C-b”,“C-f”,“C-a”,“C-e”
コマンド・ファイル名の補完	[Tab]
標準入力のリダイレクト (コマンドの入力をファイルから与える)	**コマンド** < **入力内容を書いたファイル名**
標準出力のリダイレクト (コマンドの出力をファイルに保存)	**コマンド** > **出力内容を書き込むファイル名**
コマンド 1 の出力を別のコマンド 2 の入力に与える	**コマンド** `1` | **コマンド** `2`
バックグラウンドで実行 (コマンドの終了を待たずに次のプロンプトを表示)	**コマンド** `&`

F Emacs のショートカット

利用方法を見る	“C-h T”
編集の終了	“C-x C-c”
カーソルの移動	カーソルキー ([←],[→],[↑],[↓]),“C-b”,“C-f”,“C-p”,“C-n”
カーソルを行の先頭に移動	“C-a”
カーソルを行の最後に移動	“C-e”
ファイルを開く	“C-x C-f”
ファイルを上書き保存	“C-x C-s”
ファイルを別名で保存	“C-x C-w”
一行切取り	(行の先頭で)“C-k”
一行コピー	(行の先頭で)“C-k C-y”
切取り・コピーした内容を貼り付け	“C-y”
日本語入力モードの切り替え	“C-o”
文節を伸ばす・縮める	“C-o”, “C-i”

付録 B

コンピュータ実習室 (GI201) 使用許可申請書

コンピュータ実習室 (GI201) 使用許可申請書

申請日 :　　年　　月　　日 (　　)

申請者
所属　　:
学籍番号　:
氏名　　:

下記の様にコンピュータ実習室 (GI201) を使用することを申請します。

申請理由　:

使用期日　:　　～
使用時間　:　　～

許可者
所属　　:
職　　:
氏名　　:　　印

警備員の方へ：
教職員の押印がなされた本書を持参した場合、上記の使用期間中はプログラミング演習室 (GI-201) を開錠して下さるようお願いします。