C言語プログラミング演習

茨城大学工学部情報工学科
新納 浩幸

shinnou@mx.ibaraki.ac.jp

目 次

はじめに

このテキストはシステム工学科 B コース 2 年生の「プログラミング演習」のテキストです．授業はこのテキストをもとに進めます．

授業中に聞いた説明だけで，そこで習った事項を使ったプログラムが組めるはずはありません．ですので，このテキストは予習用だと考えておいてください．授業はこのテキストに沿っているので，次週に習う内容を予習をしておくことは可能です．

またこのテキストはとりあえずプログラムが組めるようになることを念頭において書きました．私の経験上，プログラムを作る上で最低必要と思える点を中心に書いているので，説明を省いている重要事項も多いと思います．また，文章もわかりづらいかもしれません．必要に応じて自分に合った参考書を併用した方がいいでしょう．

現在，理系のどのような分野に進んでも，コンピュータは重要な役割を担っています．理系の人間にとって，プログラムできることは大事です．頑張ってください．

第1章　C 言語を学ぶ前に知っておくこと

C 言語のプログラムを作る場合，以下の手順を踏みます．

1. プログラムを書く．
2. コンパイルする．
3. プログラムを実行する．

これら３つのステップは C 言語の文法とは別個の話ですが，この３つのステップを具体的にどのように行うかを知らなければ，プログラムは作れません．この章ではこれら３つのステップについて説明します．

しかし，これら３つのステップの具体的な実行方法は，利用する OS や C 言語のコンパイラに依存します．そして上記の３つのステップを理解するためには，多少の OS の知識を必要とします．ここでは OS として Windows XP 上の DOS を，また C 言語のコンパイラとしては Borland C++ Compiler 5.5[1]を想定しています．

上記の３ステップを説明する前に，まず最低限必要とされる DOS の基礎知識を説明します．

1.1　Windows と DOS

計算機システムを使いやすく，有効に活用できるように用意されているソフトウエア体系を OS (Operating System) といいます．ハードとの橋渡しやジョブの管理などを行います．現在の計算機を使う上では必ず１つだけ何らかの OS を使っています．現在最も普及している OS は Windows XP です．Windows 98, Me や 2000 もまだ利用されていますが，ここで使う DOS 環境に限れば，大差ありません．ここでは，これら OS を総称して Windows と呼ぶことにします．

一般のユーザは OS をあまり意識することはありません．そのために，誤解している点も多くあります．ここで OS について注意すべき点を列挙してみます．

- 現在は Windows が主流．しかし OS は他にもたくさんある．
- １つの計算機に載せることのできる OS は１つとは限らない．複数の OS を搭載している計算機は，一般に，立ち上げ時にどの OS を起動するかを指定する．
- DOS も OS の一つ． Windows の出現以前は主流であった．
- DOS にもいろいろ種類があるが，一般にはマイクロソフトが作った MS-DOS のことを指す．
- DOS は UNIX という OS を模倣して作られた．
- Windows 内で動く DOS は OS というよりも，OS のエミュレータという位置づけ．

[1] C++ というプログラム言語は C 言語とは異なる言語です．ただし C++ は C の上位互換の言語となっています．そのために C++ のコンパイラは C 言語で書かれたプログラムをコンパイルすることもできます．

このテキストの学習者が利用している OS は Windows を想定しています．C 言語で作るプログラムは何らかの OS の上で実行されることが想定されています．そしてここで想定されている OS は Windows ではなくて DOS です．この話は少し複雑です．プログラムの開発は Windows 上で行いますが，プログラムは DOS 上で実行します．なぜこのようなややこしいことになるのでしょうか？

理由はいろいろあるとおもいますが，私は「初心者がいきなり Windows 上のプログラムを作ることは，プログラムの学習という観点からは適していない」ことが最大の理由だと思っています．Windows 上で，あるウインドウを開いて，そこに画像を表示させたり，それを動かしたりするプログラムはとても複雑です．現在はそのようなプログラムを一から作ることはありません．雛形の部分が提供され，足りない部分を付け足すような形でプログラムを行います．これは基本的なプログラムを作れる能力があるという前提で，行える作業です．ですので，DOS 上でのキャラクターベースのプログラムを最初に学ぶのがよいと思います．

DOS 上で実行されるプログラムを作るには，当然，計算機の OS が DOS である必要があります．ただし OS が Windows である場合には，通称「DOS 窓」というもので DOS 環境を得ることができます．図 1.1 は「DOS 窓」です[2]．この「DOS 窓」内の環境は計算機の OS が DOS である環境をエミュレートしているので，この「DOS 窓」内でプログラムのコンパイルや実行を行うことができます．

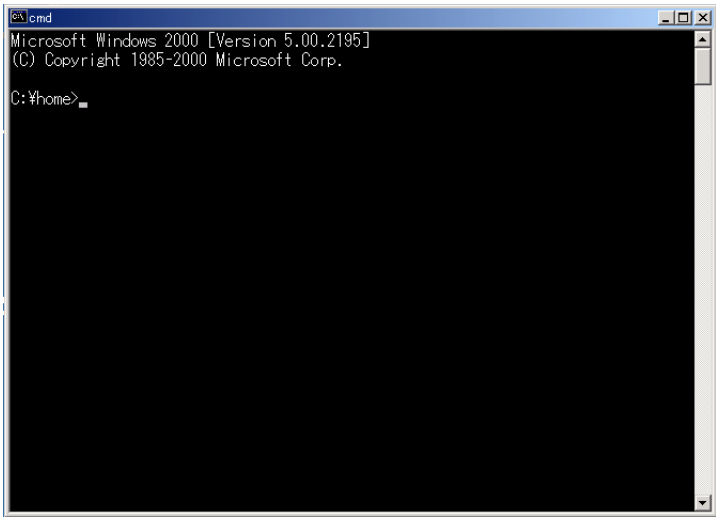

図 1.1: DOS 窓

1.2 木構造を持つファイル管理システム

多くの OS は木構造を持つファイル管理システムを採用しています．そして Windows や DOS もそうです．このファイル管理システムを理解するには，まず**ファイル**と**ディレクトリ**の理解が必要です．正確な定義を与えるのは難しいので，ここでは概念的な話にとどめますが，この程度の理解でも困ることはありません．

[2] これは Windows 2000 の DOS 窓です．

まずファイルは紙のようなもの，ディレクトリはファイルを入れる袋のようなものと捉えておけば良いでしょう．フィイルにはいろいろな種類のものがありますが，ひとまず**実行ファイル**と**テキストファイル**だけ知っていれば良いと思います．実行ファイルについては後述します．テキストファイルは人間が読める文字だけが書かれた紙と捉えていて下さい．ディレクトリはファイルを入れる袋ですが，この袋にはファイルの他にディレクトリも入れることができます．つまり袋の中にはまた袋がある場合もあります．

コンピュータシステム全体はファイルとディレクトリから構成されています．そのためすべてのファイルとディレクトリの配置関係を記述することができます．そして，その配置関係は木構造として捉えるとすっきりします．一番外側の大きな袋を木の根（**ルートディレクトリ**と呼ばれる）と考え，その袋に入っているファイルやディレクトリをその根から出ている葉と考えます．ディレクトリにあたるノードからは，さらにその中身のファイルやディレクトリの葉が延びています（図 1.2 参照）．

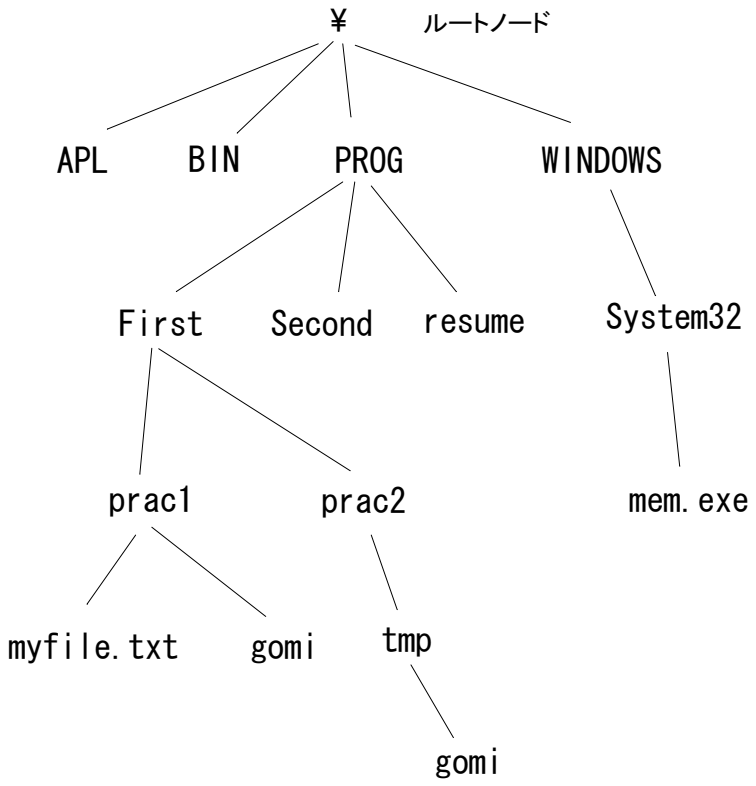

図 1.2: ファイルのツリー構造

1.3 絶対パスと相対パス

DOS を使う上では，必ず，現在自分はどこかのディレクトリにいます．そこが作業している場所です．このディレクトリを**カレントディレクトリ**といいます．

あるファイルを指定するには，**絶対パス**と**相対パス**の 2 通りの表し方があります．

絶対パスはそのファイルをルートノードからディレクトリをバックスラッシュ \ で区切って表したものです[3]．例えば，図 1.2 の myfile.txt というファイルは以下で表せます．

```
\prog\first\prac1\myfile.txt
```

相対パスとは，そのファイルがカレントディレクトリから相対的にどういう位置にあるかを示したものです．例えば，カレントディレクトリが図 1.2 の prac2 だとします．myfile.txt というファイルは以下で表せます．

```
..\prac1\myfile.txt
```

ここで .. はカレントディレクトリの一つ上のディレクトリを表します．また . はカレントディレクトリ自身を表します．
カレントディレクトリが prac2 だとします．絶対パスで

```
\prog\first\prac2\tmp\gomi
```

というファイルは，相対パスで書くと，以下のようになります．

```
tmp\gomi
```

また

```
\prog\resume
```

というファイルは，相対パスで書くと，以下のようになります．

```
..\..\resume
```

1.4 コマンドは実行ファイル

実行ファイルは計算機に何らかの動作を行わせるファイルです．その動作を実際に行わせるためには，その実行ファイルを指定して，リターンキーを打ちます．実行ファイルの指定方法も絶対パスと相対パスの方法があります．例えば \Windows\system32\mem.exe は実行ファイルであり，現在の計算機のメモリ状況を表示します．この実行ファイルを実行するには，以下のようにします．

```
> \Windows\system32\mem.exe    Enter
```

上記は絶対パスの方法です，もしも現在のカレントディレクトリが \Windows だとすれば，相対パスを利用して，以下のように実行することもできます．

```
> system32\mem.exe    Enter
```

また実行ファイルのファイル名の拡張子は .exe あるいは .com である場合がほとんどです．これらの拡張子は省略できます．つまり以下のような形でも実行できます．

[3]バックスラッシュは，環境によっては円記号の ¥ と表示される．どちらもコンピュータ内部では同じコードです．表示だけが異なります．

```
> \Windows\system32\mem      Enter
```

あるいは

```
> system32\mem      Enter
```

1.5 パス変数と内部コマンド

よく使うコマンドの場合は，いちいちそのファイルのある場所を指定するのはめんどうなので，**パス変数**というもので，コマンドを探しにゆくディレクトリを予め複数指定しておくことができます．パス変数で指定されたディレクトリの下にある実行ファイルは，位置を指定しなくてもコマンド名（ファイル名）だけで実行できます．

パス変数は PATH で確認できます．

前章でコマンドとは実行ファイルであることを述べました．しかしそれ以外のコマンドも存在します．例えば dir はカレントディレクトリ下にあるファイルとディレクトリを表示するコマンドです．このコマンドは位置を指定せずに，直接以下のように実行できます．

```
> dir      Enter
```

このため，一見，パス変数に設定したディレクトリの下に， dir.exe あるいは dir.com といったファイルが存在するようにも思えますが，そのようなファイルはありません．このように実行ファイルとしての形をもっていないコマンドを**内部コマンド**といいます．内部コマンドの実体はDOS そのものが持っているコマンドです．そのため，基本的なコマンドの多くは内部コマンドとなっています．

1.6 DOS の基本コマンド

いくつかのコマンドは DOS 窓でファイルの操作を行う上で重要です．最低以下のコマンドは使えるようにすべきです．

● dir

基本的には以下のように実行して，カレントディレクトリにあるファイルとディレクトリを表示させます．

```
> dir      Enter
```

また，ディレクトリ名を指定すると，指定されたディレクトリの下にあるファイルとディレクトリが表示されます．ディレクトリの指定方法は，ファイルと同様，絶対パスによる方法と，相対パスによる方法があります．以下はルートディレクトリの下にある Windows というディレクトリを指定して，dir を実行する例です．

```
> dir  \windows   Enter
```

● cd （ディレクトリ名）

指定されたディレクトリに移ります．つまりカレントディレクトリを指定されたディレクトリに変更します．以下ではディレクトリ `\windows\system32` に移ります．

```
> cd  \windows\system32   Enter
```

またディレクトリを指定せずに cd を実行すれば，カレントディレクトリが表示されます．

● md （ディレクトリ名）

指定したディレクトリを作成します．

● copy （ファイル名１）（ファイル名２）

ファイル名１で指定されたファイルの内容をファイル名２で指定されたファイルにコピーします．
ファイル名２が既に存在してるファイルの場合，ファイル名２の中身がファイル名１に変更され，先のファイル名２の中身は消えるので注意して下さい．

● ren （ファイル名１）（ファイル名２）

ファイル名１をファイル名２に変更します．つまりファイル名を変更します．ファイルの中身は変化しません．

● del （ファイル名）

ファイル名で指定されたファイルを削除します．

● type （ファイル名）

ファイル名で指定されたファイルの中身を表示します．ただし大きなファイルだと一画面に表示できず，どんどんスクロールしてゆきます．一画面づつゆっくり見たい時は， `more` を使います．以下のようにして使います．SPACE で次の画面が表示されます．

```
> more  bigfile.txt  Enter
```

上記では各コマンドの基本的な使い方を示しましたが，コマンドにはいろいろなオプションをつけて，細かい処理の制御が可能です．例えば，

```
> dir  /w   Enter
```

とすれば，ファイル名とディレクトリ名だけの表示になり，たくさんのファイルをもつディレクトリの中でファイルを探すときに便利です．
各コマンドがどのようなオプションをもつかは解説書を読まないと分りません．ただし，

```
> dir  /?   Enter
```

というように，/? というオプションをつけることで，そのコマンドにどのようなオプションがあるかを調べることができる場合もあります．

1.7 プログラムを書く

C 言語のプログラムはテキストファイルです．テキストファイルとは何かをここでは説明しません．正確な定義はコンピュータの細かい知識が必要です．おおざっぱに人間が読める文字が書かれた紙のようなものと考えておいて問題はありません．プログラムを書くとは，C 言語の文法に従って計算機に与える命令の手順を記述したテキストファイルを作成すること です．C 言語の文法は次章以降に学習します．ここではどのようにしてテキストファイルを作成するかを述べます．

結論から述べると，テキストファイルを作成するのは，**エディタ**と呼ばれるアプリケーションを利用します．また利用できるエディタは様々なものが多数存在します．Windows で標準についている「メモ帳」も一種のエディタです．自分が使いやすいエディタを選び，そのエディタを利用してテキストファイルを作成します．具体的にどのようにしてテキストファイルを作成するかは，利用するエディタに依存した話なので，ここでは説明しません．

結局，具体的な作業方法については何も述べていませんが，テキストファイルを作成するのはエディタであることを理解して下さい．プログラムを生業とする人が最も利用するソフトはエディタです．Internet Explorer や Excel まして Word などでは決してありません．エディタの操作に習熟すべきです．それがプログラムの生産性を向上させます．

どんなエディタを使えば良いか．もし自分で特に気に入ったものがないのであれば，Emacs を使うことを勧めます．私は Emacs を越えるエディタは存在しないし，今後もあらわれないと思っています．Emacs は UNIX 上のエディタでしたが，現在は Windows 等の他の OS にも移植されています．Windows では Meadow と呼ばれています．以下のサイトからダウンロードできます．

```
http://www.vector.co.jp/soft/win95/writing/se101233.html
```

1.8 コンパイルする

プログラム言語は一般に**コンパイラ言語**と**インタプリタ言語**に大別できます．コンパイラ言語のプログラムでは，プログラムをコンパイルすることで，実行形式のファイルを作成します．その実行形式のファイルを実行することで，計算機に所望の計算を行わせます．インタプリタ言語のプログラムは，インタプリタと呼ばれるアプリケーションによって，順次解釈されて，計算機に所望の計算を行わせます．コンパイラ言語の例としては，C ，C++ ，Fortran などがあります．インタプリタ言語の例としては，Lisp ，Perl などがあります．

C 言語はコンパイラ言語です．そのためプログラムを書いただけでは，そのプログラムを実行することはできません．書いたプログラムをコンパイルして実行形式のファイルを作成する必要があります．

コンパイルするソフトは**コンパイラ**と呼ばれます．C 言語のコンパイラも多数あります．有償ではマイクソフトの Visual C++ が有名です[4]．具体的にどのようにしてコンパイルするかは，利用するコンパイラに依存します．ここでは Borland C++ Compiler 5.5 を例にして説明します．

ただし　Borland C++ Compiler 5.5 のインストール方法については述べません．利用に関してのポイントだけ述べます．ここでは Borland C++ Compiler 5.5 が `c:\Borland\BCC55` というディレクトリ下にインストールされたと仮定します．コンパイルする前に，PATH 変数に `c:\Borland\BCC55\Bin` を追加します．また `c:\Borland\BCC55\Bin` の下に以下の内容を記述した `BCC32.CFG` というファイルを作成します．

[4]先にも述べましたが，C++ のコンパイラは C のコンパイラとしても機能します．

```
-IC:\Borland\BCC55\Include
-LC:\Borland\BCC55\Lib
```

同様に，c:\Borland\BCC55\Bin の下に以下の内容を記述した ILINK32.CFG というファイルを作成します．

```
-LC:\Borland\BCC55\Lib
```

以上の設定ができれば，自分が書いたプログラム myprog.c は以下のようにしてコンパイルすることができます．bcc32.exe がコンパイラです．

```
> bcc32  myprog.c    Enter
```

コンパイルが正常に終了すれば，カレントディレクトリに， myprog.exe という実行ファイルが作成されています．

1.9 プログラムを実行する

コンパイルして作成できた実行ファイルを実行するのは，DOS 上で，その実行ファイルを指定してリターンキーを押すだけです．指定の方法は既に学びました．

演習問題 1

1-1 図 1.2 でカレントディレクトリが \PROG\first\prac1 であるとします．このディレクトリから移動せずに，mem を実行する方法として，絶対パスを利用する方法と相対パスを利用する方法を書きなさい．

1-2 また先と同様，上記のカレントディレクトリを移動せずに\PROG\first\prac2\tmp\gomi というファイルを消す方法を絶対パスを利用する方法と，相対パスを利用する方法の 2 通りで書きなさい．

1-3 DOS 窓を起動した後に，自分の PATH がどのように設定されているかを調べなさい．

1-4 copy コマンドで使えるパラメータを調べなさい．

第2章　画面への出力と変数の宣言

2.1　プログラムの骨格

C のプログラムは，ほとんどの場合，以下の形をしています．これがプログラムの骨格です．厳密に書くと，もう少し色々必要ですが，しばらくはこれで話を進めます．この骨格の部分はどのようなプログラムを作るときにでも，必要になるので，どこかのファイルに書いておき，何かプログラムを書く時はそのファイルをコピーして，コピーしたファイルを編集すると楽です．

```
#include <stdio.h>

int main()
{

}
```

最後の { } の中にやりたいことを書きます．やりたいことは文の並びで表します．文とは何か？ C 言語の命令だと考えて下さい．文についての注意として，

文の最後には ； をつける

とひとまず覚えておきましょう．

また本書では，プログラムを例示する際に， `#include <stdio.h>` の部分を省略することもあります．適時補って理解してください．

また以下の形をプログラムの骨格とする人もいます．

```
#include <stdio.h>

int main()
{

   return 0;
}
```

この形の方が正統です．ただし main が返す値に意味が生じるようなプログラムでなければ，`return 0` は必要ありません．

2.2 文と中括弧

C 言語のプログラムは**文**の集まりです．1 文づつ，上から順に実行されてゆきます．

複数の文をまとめて，1 文として扱いたいときもあります．そのとき，中括弧{ }を使います．中括弧{ }の中には文が並んでいます．つまり文は以下の形のいずれかです．

○○○○○；

か

{ ○○○○○；○○○○○； … ○○○○○； }

中括弧{ }の中の文は全体で 1 つの文として扱われます．その全体の文の実行は，またその中の文が 1 文づつ上から順に実行されてゆきます．

2.3 画面に文字を出す

プログラムで絶対に最初に覚えなければならないことは，画面に文字を出す方法です．これができないと，プログラムが正しく動いたのかを確認できません．いろいろな方法がありますが，printf という関数を使うのが最も簡単で強力です．以下に printf の使用例をいくつかあげてみます．

練習 2-1 以下の内容を myself.c というファイルで保存し，コンパイルして myself.exe を作り，myself を実行しなさい．

```
int main()
{
     printf("私は茨城大学の学生です\n");
}
```

[実行結果]

私は茨城大学の学生です

printf の基本の使い方は，画面に出力したい文字列を二重引用符 (") で囲み，それを printf の引数にすることです．その文字列を画面に出した後に改行したければ文字列の最後に \n を入れます．\n がないとどうなるのか，以下の 2 つのプラグラムを作って確認してみて下さい．

```
/* prog1 */
int main()
{
     printf("茨城\n");
     printf("大学\n");
}
```

```
/* prog2 */
int main()
{
     printf("茨城");
     printf("大学\n");
}
```

prog1 の出力は

茨城
大学

prog2 の出力は

茨城大学

となります.

練習 2-2 以下のような文章を画面に出すプログラムを作りなさい.

茨城大学工学部は日立市にあります.
いろいろ長所もあるんですけど，なんとなく
予備校みたいなイメージを持ってしまいます.
もっとサークル活動などが活発だといいと思います.

以下は答えの一例です．当然，答えは無数にあります.

```
int main()
{
   printf("茨城大学工学部は日立市にあります．\n");
   printf("いろいろ長所もあるんですけど，なんとなく\n");
   printf("予備校みたいなイメージを持ってしまいます．\n");
   printf("もっとサークル活動などが活発だといいと思います．\n");
}
```

2.4 コメント

プログラムを作った当初は，そのプログラムの中身はよく理解できますが，しばらくするとその中身を理解するのは書いた本人でさえ難しくなることがよくあります．そのために，プログラムに説明のためのメモなどを入れておくのがよいとされています．そのようなメモを**コメント**といいます.

コメントの入れ方の基本は，そのコメントを /* と */で囲みます．その中の文章がコメントアウトされます．コメントアウトされた部分は，コンパイラは読まないので，コンピュータにとっては，コメントはあってもなくても違いはありません．しかし先に述べたように人間にとっては大事です．できるだけコメントは入れましょう．ただし程度問題もあります．あまり明らかなことにいちいちコメントをつけるのは，逆に読みづらくなります.

コメントの中にコメントを入れるとどうなるかはコンパイラに依存しますので，一般に行ってはいけません．

```
int main()
{
   /*  この部分は注釈，何でも書ける．  */

   /*  ２行にわたって書く時でも，
       ＯＫです．
   */
   printf("これは出力されるよ．\n");   /* こんな場所にもかけるのさ */
   /*
      printf("これは注釈の中だから出力されない．\n");
   */
}
```

上記のプラグラムの実行結果は以下の通りです．

［実行結果］

これは出力されるよ．

コメントの入れ方として // を使う方法もあります．// から改行までの部分がコメントアウトされます．コメントが１行ならこの書き方も便利です．ただ // は古い C コンパイラーでは実装されていないこともあるので，/* と */で囲む方法を使うことを勧めます．

```
int main()
{
   //  この部分は１行の注釈，// を使うと簡単

   //  ２行にわたって書く時は，// を
   //  ２回つかうとよい

   printf("これは出力されるよ．\n");   // こんな場所にはこのように使う

   //   printf("これは注釈の中だから出力されない．\n");

}
```

［実行結果］

これは出力されるよ.

2.5 変数を持つプログラム

プログラム上の**変数**は，数学の場合の変数と考え方がほとんど同じです．四則演算（足し算，引き算，かけ算，割算）は自然に書けば実現できます．ただし使う記号は

足し算（+），引き算（-），かけ算（*），わり算（/），

です．また割算の場合，整数どうしの割算は整数までしか求まらない，つまり切り捨てされることにひとまず注意しておいて下さい（型変換の規則を参照）.

例) `7 / 3`　（この式の結果は 2 余り 1 → この式の結果は 2 になる）

また**除余**（%）という演算もあります．これは割り算の余りが求まります.

例) `7 % 3`　（この式の結果は 2 余り 1 → この式の結果は 1 になる）

カッコは数学と同じ意味で使えます．計算の順番が不明瞭になる場合は，不必要でもカッコを入れた方が読みやすくなります.

例) `1 + 2 * 2 + 3`　（この式の結果は 8 になる）

例) `(1 + 2) * (2 + 3)`　（この式の結果は 15 になる）

また C 言語の等号（=）は代入を意味します．これは数学とは異なるので注意が必要です．例えば，以下の式は変数 `x` の値に 3 を加え，それを変数 `y` に代入するという意味です.

```
y = x + 3;
```

上記の例は通常の等号の意味で捉えても不都合はありませんが，以下の例では代入という意味で取らないと理解できません.

```
x = x + 3;
```

上の式は変数 `x` の値に 3 を加え，それを変数 `x` に代入するという意味です．つまり，この式を実行することで `x` の値は変更されます.

練習 2-3 1 から 10 まで足してその和を表示するプログラムを書きなさい.

変数の練習として，以下のような答も書いておきます.

```
int main()
{
   int  sum;  /* これは変数の型宣言，後述 */
   sum = 0;
   sum = sum + 1;
   sum = sum + 2;
   sum = sum + 3;
   sum = sum + 4;
   sum = sum + 5;
   sum = sum + 6;
   sum = sum + 7;
   sum = sum + 8;
   sum = sum + 9;
   sum = sum + 10;
   printf("1から10までの和は %d  だよ\n",sum);
}
```

[実行結果]

```
        1から10までの和は 55  だよ
```

普通は以下のように書きます.

```
int main()
{
   int  sum;   /* これは変数の型宣言，後述 */
   sum = 1+2+3+4+5+6+7+8+9+10;
   printf("1から10までの和は %d  だよ\n",sum);
}
```

本当は，変数は必要ありません.

```
int main()
{
   printf("1から10までの和は %d  だよ\n",1+2+3+4+5+6+7+8+9+10);
}
```

変数を１増加させる，あるいは１減少させるという処理はよく行われます．

```
i = i + 1;    や   i = i - 1;
```

これらに対して，以下の特別な演算子があります．これらを使った方が分りやすいです．

```
i = i + 1;    ==>   i++;    （インクリメンタル演算子）
i = i - 1;    ==>   i--;    （デクリメンタル演算子）
```

似たような省略形の ++i や --i もあります．これらは単独の一文で使われる限り， i++ や i-- とは同じですが，その文の返り値を使おうとする場合に仕様が異なります．

++i や i++ の返り値を使うのは，間違いやすいのでやらない方がよいと思います．

省略形は１の増減だけではありません．例えば，i = i + a は i += a と書けます．同様に，以下の関係もあります．

```
i = i + a;    ==>   i += a;
i = i - a;    ==>   i -= a;
i = i * a;    ==>   i *= a;
i = i / a;    ==>   i /= a;
```

以下に変数についての注意点をまとめてみます．

- 変数の名前はアルファベット文字列ならほぼ自由に使えますが，使ってはならない名前（**予約語**）もあります．どのような予約語があるかは， あまり気にしなくてもかまいません．概ね，C 言語で予め使われている単語です．
- 変数を使う場合には，変数がどのような種類（**型**）の値が入るのかを宣言しなくてはなりません（**変数の型宣言**）．宣言された以外の型の値は基本的には代入できません．宣言は以下のような文で行います．

```
型の名前    変数の名前 ;    /*  これが宣言 プログラムの最初に書く*/
```

基本的な型として以下の４つを最低覚えて下さい．

整数型 変数の値が整数のときの型．値の例としては 1 や -12 などがあります．

```
int  a;
```

倍精度実数型 変数の値が実数であるときの型．単精度実数型という型もありますが，特別な理由がない限り，実数は倍精度実数型で宣言するのが普通です．値の例としては 12.1 や -23.67 などがあります．

```
double  a;
```

文字型 変数の値が半角英数字 1 文字のときの型．ここでは日本語文字のような全角文字は対象外です．注意してください．また半角カタカナ文字も対象外と考えて下さい．値はシングルクオート（引用符）' で囲みます．値の例としては 'a' や '8' などがあります．

```
char  a;
```

文字列型 文字の連続した並びが文字列です．ただ C 言語には文字列型という型はありません．しかし文字列は頻繁に利用します．その実現方法は後で学習します．とりあえず，単独の文字列の変数は以下のように宣言して使うことを覚えておけばよいでしょう．値はダブルクオート（２重引用符）"で囲みます．文字列の場合には，何か特別な処理をしない限り，全角文字が入っても問題ありません．

```
char* a = "茨城大学";
```

上記の使い方は注意が必要です．文字列については後の章で説明するので，それまでは文字列を出力する以外の用途では使用しないで下さい．

- 変数はまとめて宣言しても，個別に宣言してもかまいません．以下に例を示します．

```
int main()
{
   int  a,b,sum;     /* <== まとめて変数を宣言している */
   a = 1+2+3+4+5;
   b = 6+7+8+9+10;
   sum = a + b;
   printf("1から 10 までの和は %d  だよ\n",sum);  /* この文の使い方は次の章 */
}
```

[実行結果]

```
1から 10 までの和は 55  だよ
```

```
int main()
{
   int  a;       /* <==  個別に宣言している */
   int  b;       /* <==  個別に宣言している */
   int  sum;     /* <==  個別に宣言している */
   a = 1+2+3+4+5;
   b = 6+7+8+9+10;
   sum = a + b;
   printf("1から 10 までの和は %d  だよ\n",sum);   /* この文の使い方は次の章 */
}
```

[実行結果]

```
1から 10 までの和は 55  だよ
```

- 宣言された変数は，何かを代入するまで，その変数の値には何が入っているか分りません．そのために何かを代入する前にその変数の値を参照したプログラムを作ると，滅茶苦茶な結果になります．そのため，変数を宣言したら，必要であれば，プログラムの最初の方でその変数の初期値を代入しておくとよいでしょう．

```
int i;
i = 5;
```

初期値の代入を宣言文の中で行うこともできます．これを**変数の初期化**といいます．例えば，上記の例は変数 i を int 型で宣言し，初期値として 5 を代入しています．これを変数の初期化で行う場合は以下のようになります．

```
int i = 5;
```

2.6 変数の出力

変数の値を出力するにも printf を使います．
printf は正確に書くと以下のようなフォーマットになっています．

```
printf(出力文字列，変数 1，変数 2，...，変数 n)
```

基本的に出力文字列の部分が出力されます．出力文字列の部分にある特別の記号が書かれていたら，その部分が，第 2 引数以下の変数の値に置き換わります．

いくつか例を示します．まず変数が１つもない場合の printf の使い方は既に学びました．例えば，以下の例では出力文字列が「私は茨城大学の学生です」＋「改行（\n)」となっています．

```
printf("私は茨城大学の学生です\n")
```

次に変数が１つある場合の例を示します．その変数 x は int 型であるとします．出力文字列中のある地点にその変数の値を出力するには，出力文字列中のその地点に %d を置き，printf の第 2 引数に，その変数 x を指定します．

例えば，以下のような文字列を出力したいとします．ただし下線部 __ の部分に変数 x の値を出力したいとします．

```
変数 x の値は __ です
```

この場合，以下のように書けばよい．

```
printf("変数 x の値は %d です", x);
```

次に変数が２つある場合の例を示します．その変数 x と y はともに int 型であるとします．出力文字列中の変数 x と変数 y を置きたい地点に %d を置きます．この場合，出力文字列中には 2 つの %d がありますが，1 番目の %d に対応する変数が x なら，printf の第 2 引数に変数 x を指定し，printf の第 3 引数に変数 y を指定します．逆に 1 番目の %d に対応する変数が y なら，printf の第 2 引数に変数 y を指定し，printf の第 3 引数に変数 x を指定します．

```
printf("変数 x の値は %d で，変数 y の値は %d です", x, y);

printf("変数 y の値は %d で，変数 x の値は %d です", y, x);
```

変数が３つ以上ある場合は，……，以上をもとに推理して容易に書けると思うので省略します．基本は，printf の第 1 引数の出力文字列中にある特別の記号が書かれていたら，その部分が，対応する第 2 引数以下の変数の値に置き換わることです．

変数の値に置き換わるある特別な記号とは何でしょう．この部分の記号には，出力される型や出力のフォーマットを指定した形になるため，細かい仕様を書くのは大変です．とりあえず，基本的なものだけ以下に示します．

```
%d      整数型の変数の値に置き換わる
```

%lf　実数型の変数の値に置き換わる（倍精度）
%f　実数型の変数の値に置き換わる（単精度）
%c　文字型の変数の値に置き換わる
%s　文字列を指す変数のその文字列に置き換わる

C には文字列型という型がありませんので，最後の %s だけは少し特殊です．文字列の節で説明するので，今は気にしなくてもかまいません．
以下に利用例を示します．

```
int main()
{
   int     a = 2;
   double  b = 3.1;
   char    c = 'z';

   printf("整数型の変数の表示 %d  \n",a);
   printf("実数型の変数の表示 %lf \n",b);
   printf("文字型の変数の表示 %c  \n",c);
   printf("同時に表示 %d, %lf, %c \n",a,b,c);
}
```

[実行結果]

```
整数型の変数の表示 2
実数型の変数の表示 3.100000
文字型の変数の表示 z
同時に表示 2, 3.100000, z
```

演習問題 2

2-1 以下の 3 行の文書を 1 つの printf で表示しなさい.

茨城大学は
茨城県の
日立市にあります

2-2 三角形の底辺と高さの数値を変数（実数型）に代入し，その三角形の面積を計算するプログラムを書きなさい.

2-3 球の半径を変数（実数型）に代入し，その球の体積を計算するプログラムを書きなさい.

2-4 printf を用いて以下の文書を出力しなさい．%d をどう出力させるかがポイントです．調べないとできません.

%d は整数の値に置換される

第3章　繰り返しと分岐

3.1　繰り返しのあるプログラム（while 文，for 文）

3.1.1　while 文

プログラムでは繰り返しの処理が本質的です．繰り返しを行なうことで，コンピュータは人間が行えない大量の計算を行うことができます．繰り返しを行うための命令文はいくつかありますが，`while` 文と `for` 文だけ知っていれば十分です．まず，最も汎用性の高い `while` 文を説明します．while 文は以下の構造を持ちます．

```
while ( XXX )
    YYY;
```

XXX の部分には条件式が入ります．この条件式が真（正しい）である限り，YYY という文を繰り返します．具体的には，まず XXX の条件を調べます．真であれば，YYY を実行します．偽であれば while 文を終了します．YYY を実行した場合，YYY の処理が終ったら，再び，XXX の条件を調べます．真であれば，再び YYY を実行します．偽であれば while 文を終了します．結局，これが繰り返されます．

while 文の本体にあたる YYY には複数の文が対応するのが普通です．なので，通常は，while 文の構造は，以下の形になっています．

```
while ( XXX ) {
   YYY-1;
   YYY-2;
    …
   YYY-n;
}
```

3.1.2　条件式

上記の XXX に入るのは**条件式**です．条件式にはいろいろありますが，以下の４つだけを知っていれば十分です．これだけでほとんどのことはできます．

- 変数 a と変数 b が等しいかどうかの条件.

```
a == b
```

（注意 ＝ が２つ！）

等しければ，真となり，等しくなければ偽となります.

- 変数 a と変数 b が異るかどうかの条件.

```
a != b
```

異っていれば，真となり，等しければ偽となります.

- 変数 a が変数 b よりも大きいかどうかの条件.

```
a > b
```

大きければ，真となり，大きくなれば偽となります.

- 変数 a が変数 b 以上であるかどうかの条件.

```
a >= b
```

以上であれば，真となり，以上でなければ偽となります.

また条件式は論理式のように，否定，論理和，論理積が使えます.

否定 条件式の前に！をつけると否定になります.

例） `!(a == b)`

括弧の中は a と b が等しければ真，等しくなれれば偽です．その否定を取るので，a と b が等しければ偽，等しくなければ真となります．結局， `a != b` と同じ意味です.

論理和 条件式と条件式の間に `||` を入れます.

例） `(a == b) || (c > d)`

a と b が等しい，あるいは，c が d より大きいならば真，そうでなければ偽.

論理積 条件式と条件式の間に `&&` を入れます.

例） `(a == b) && (c > d)`

a と b が等しい，かつ，c が d より大きいならば真，そうでなければ偽.

練習 3-1 「私は茨大生です」という文を 100 回出力するプログラムを作りなさい．

この問題に対するプログラムとして，以下のプログラムが最も原始的なプログラムです．ただし，これだとプログラムを書く意味がありません．繰返しの命令を使うべきです．

```
int main()
{
    printf("私は茨大生です\n");    /*  1 回目  */
    printf("私は茨大生です\n");    /*  2 回目  */
    printf("私は茨大生です\n");    /*  3 回目  */

    以下延々とこれを書いてゆく

    printf("私は茨大生です\n");    /*  100 回目  */
}
```

[実行結果]

```
私は茨大生です
私は茨大生です
私は茨大生です
...
私は茨大生です
```

繰り返しの命令を使った答が以下です．

```
int main()
{
    int i;

    i = 0;    /* 変数 i は何回表示したかを表す整数．最初は０ */
    while(i < 100) {
       printf("私は茨大生です\n");
       i++;
    }
}
```

3.1.3 変数をもつ繰り返し

先のプログラムは実際に繰り返したい命令が全く同じでした．通常の繰り返しでは繰返す命令が，ほとんど同じだけど少しだけ違うという形になります．この違いは一般に変数で実現されます．

練習 3-2 以下のような文を表示するプログラムを作りなさい．

```
私は茨大生です．あと 99 回言うぞ．
私は茨大生です．あと 98 回言うぞ．
私は茨大生です．あと 97 回言うぞ．
....
私は茨大生です．あと  1 回言うぞ．
私は茨大生です．あと  0 回言うぞ．
```

各行で出力される文字列は「あと○○回言うぞ．」の○○の数値だけが異なります．そして○○の数値は 99 から始まり，1 づつ減ってゆきます．この関係を考えると，k 回目に出力される○○の数値は，`100 - k` となっていることが分ります．結局，答えは以下のようになります．

```
int main()
{
    int k = 1,i;
    while ( k <= 100 )  {
        i = 100 - k;
        printf("私は茨大生です．あと %d 回言うぞ．\n", i);
        k++;
    }
}
```

上記のプログラムにはいくつか冗長な点があります．そのひとつが，変数 `i` の使用です．上記のプログラムの `printf` の部分は，

```
printf("私は茨大生です．あと %d 回言うぞ．\n", 100 - k);
```

という形でも書くことができるので，変数 `i` を導入する必要はなかったのです．変数を使った式の値だけが必要である場合には，その式の値を新たな変数に代入することをせずに，その式自体を変数のように使う方が簡単です．

3.1.4 for 文

繰り返しの命令は while 文だけでも十分です．しかし for 文もよく使うので，これも使えるようにして下さい．

for 文は以下の while 文を

```
<初期設定>
while( <条件> )
{
   <処理>
   <処理の最後にやる命令>
}
```

以下のように変形したものです．

```
for( <初期設定> ; <条件>  ;<処理の最後にやる命令> )
{
    <処理>
}
```

例えば，練習 3-2 のプログラムは

```
int main()
{
    int i;

    for(i = 0; i < 100 ; i++)
    {
       printf("私は茨大生です．あと %d 回言うぞ．\n", 99 - i);
    }
}
```

のように書けます．for 文の初期設定は書かなくてもいいので，上の例で変数 i を宣言時に 0 に初期化して，以下のようにも書けます．

```
int main()
{
    int i = 0;    /* ここで初期設定する */

    for(     ; i < 100 ; i++)
    {
       printf("私は茨大生です．あと %d 回言うぞ．\n", 99 - i);
    }
}
```

このように，整数 i を 1 増やしながら繰り返すといったような，単純な繰り返しは for 文を使う方が簡単です．

3.2 分岐のあるプログラム (if 文)

分岐のあるプログラムとは，ある場合にはこれこれの文を実行してまたある場合にはこれこれの文を実行する，ということを行なうプログラムです．基本的に if 文を使います．ひとまず if 文だけで十分です．

if 文は以下の構造のどちらかを持ちます．

（1）

```
if ( XXX ) {
   YYY
}
```

（2）

```
if ( XXX ) {
   YYY
} else {
   ZZZ
}
```

（1）も（2）も XXX の部分は条件式です．while 文のところで説明した条件式が入ります．（1）の場合，XXX の条件式が真ならば，YYY を実行します．偽なら何もしません．（2）の場合，XXX の条件式が真ならば，YYY を実行します．偽なら ZZZ を実行します．

練習 3-3 「私は茨大生です」を 100 回表示するプログラムを書きなさい．ただし，5 回表示する毎に,「くどいです」を表示しなさい．

```
int main()
{
    int i = 0, amari;

    while ( i < 100 )  {
        printf("私は茨大生です\n");
        i++;
        amari = i % 5;
        if (amari == 0) {
           printf("くどいです\n");
        }
    }
}
```

[実行結果]

```
私は茨大生です
私は茨大生です
私は茨大生です
私は茨大生です
私は茨大生です
くどいです
私は茨大生です
...
私は茨大生です
くどいです
```

上記の書き方は少し冗長です．以下のようにも書けます．

```
int main()
{
   int i;

   for(i = 1;i <= 100;i++) {
      printf("私は茨大生です\n");
      if (!(i % 5)) printf("くどいです\n");
   }
}
```

演習問題３

3-1 4 桁の整数の中で，各桁の数字の和が 10 になる整数をすべて表示するプログラムを書きなさい．

例) 1563 という数字の各桁の和は 1+5+6+3 = 15

3-2 ある年がうるう年かどうかを求めるプログラムを書きなさい．うるう年は 4 で割りきれるが，100 では割りきれない，ただし 400 では割りきれても良い，という年です．

3-3 正の整数のうち，２桁の整数と２桁の整数の積になっているものをすべて表示するプログラムを書きなさい．

例えば，231 は 11 と 21 の積になっているので，

```
231 = 11 * 21
```

という具合いに表示するプログラムです．

第4章　繰り返しの制御

4.1　繰り返しの入れ子

繰り返しは入れ子の構造をもつことができます．繰り返しの入れ子とは繰り返しの処理の中に繰り返しが入ることです．

練習 4-1 九九の掛け算を出力するプログラムを書きなさい．

```
int main()
{
   int i,j;

   for(i = 1;i <= 9;i++) {
      for(j = 1;j <= 9;j++) {
        printf(" %2d",i*j);     /* %2d は 2 桁で表示 */
      }
      printf("\n");
   }
}
```

［実行結果］

```
 1  2  3  4  5  6  7  8  9
 2  4  6  8 10 12 14 16 18
 3  6  9 12 15 18 21 24 27
 4  8 12 16 20 24 28 32 36
 5 10 15 20 25 30 35 40 45
 6 12 18 24 30 36 42 48 54
 7 14 21 28 35 42 49 56 63
 8 16 24 32 40 48 56 64 72
 9 18 27 36 45 54 63 72 81
```

上記のプログラムは for 文の中に for 文が入っています．外側の for 文は変数 i で繰り返し，内側の for 文は変数 j で繰り返します．内側の for 文の中で変数 i はその時点の i で固定されている点に注意してください．

また上の例題には printf 文の中に %2d という記号を使っています．これは整数の出力を 2 桁にすることを意味しています．このように printf 文の中の %〜 の部分には様々な出力フォーマットを指定することができます．

4.2 繰り返しの途中終了（break 文）

繰り返しの中でその繰り返しを途中で終了したいときがあります．このような場合，break 文を使います．繰り返しが入れ子になっている場合は注意してください．break 文は最も内側の繰り返しを抜けます．

練習 4-2 九九の掛け算を出力するプログラムを書きなさい．ただし， i × j で i ≧ j となるものだけを表示しなさい．

```
int main()
{
   int i,j;

   for(i = 1;i <= 9;i++) {
      for(j = 1;j <= 9;j++) {
         if (i < j) break;
         printf(" %2d",i*j);
      }
      printf("\n");
   }
}
```

[実行結果]

```
 1
 2  4
 3  6  9
 4  8 12 16
 5 10 15 20 25
 6 12 18 24 30 36
 7 14 21 28 35 42 49
 8 16 24 32 40 48 56 64
 9 18 27 36 45 54 63 72 81
```

i ≧ j の条件の否定は i < j であることに注意すること． i < j の条件式が満たされたときに，内側の繰り返しを抜けます．

break 文を使わないプログラムは以下のようになります．上の方がすっきりしていると思います．

```
int main()
{
   int i,j;

   for(i = 1;i <= 9;i++) {
      j = 1;
      while((i >= j) && (j <= 9)) {
        printf(" %2d",i*j);
        j++;
      }
      printf("\n");
   }
}
```

4.3 次の繰り返しへの強制移行（continue 文）

例えば変数 i を増やしながら，繰り返しの処理を行っているとします．ある条件が満たされたときに，その i についての処理はもう必要なくなり，次の i に処理を移したい場合があります．そのようなときは continue 文を使います．

練習 4-3 九九の掛け算を出力するプログラムを書きなさい．ただし， i × j で i ≦ j となるものだけを表示しなさい．

```
int main()
{
   int i,j;

   for(i = 1;i <= 9;i++) {
      for(j = 1;j <= 9;j++) {
        if (i > j) continue;
        printf(" %2d",i*j);
      }
      printf("\n");
   }
}
```

[実行結果]

```
 1  2  3  4  5  6  7  8  9
 4  6  8 10 12 14 16 18
 9 12 15 18 21 24 27
16 20 24 28 32 36
25 30 35 40 45
36 42 48 54
49 56 63
64 72
81
```

この問題の場合， i ≦ j を満たさない，つまり i > j を満たす場合は，i × j を出力せずに，次の j に移りばよいです．このために continue 文を使っています．

4.4 プログラムの途中終了

繰り返しの制御とは関係ありませんが，プログラムの途中終了も繰り返しの中で使うことがあります．

基本的にプログラムを抜けたい場所に exit(0) という文を置くだけです．ただし，exit という関数は， stdlib.h に定義されているので，exit を使う場合には， stdio.h の他に stdlib.h もインクルードしておく必要があります．ここらはヘッダーファイルとライブラリのところで説明するので，とりあえず，そうするものと覚えておいてください．

練習 4-4 3401 が素数かどうかを判定するプログラムを書きなさい．

素数というのは，その数を割りきれる数が，1 とその数以外にはない数です．2,3,5,7,11，…などです．素数を判定する最も原始的な方法は，2 からその数マイナス 1 までの各々の数でその数を割ってみることです．割りきれる数があれば，その数は素数ではなく，2 からその数マイナス 1 までのすべての数で，その数を割りきることができなければ，その数は素数です．

問題は 2 からその数マイナス 1 までの各々の数で割り，途中で割り切れる数が存在して，素数でないことが分ったら，それ以上，その他の数で割る検査は必要なく，直ちにプログラムを終了させます．その命令として exit(0) を使います．

```
#include <stdio.h>
#include <stdlib.h>  /* とりあえず，これが必要と覚えておく */

int main()
{
    int i;

    for(i = 2;i < 3401;i++) {
        if (3401 % i == 0) {
           printf("素数ではない．約数として %d がある\n", i);
           exit(0);  /* 約数が 1 つ見つかったので，
                        これ以上処理を行っても意味ないので，
                        プログラムを終了させる */
        }
    }
    printf("素数でした\n");
}
```

[実行結果]

素数ではない．約数として 19 がある

実は main も一つの関数だと分っていれば，関数を途中で終了させる return という文も使えます．使い方は exit(0) の部分を return 0 と書くだけです．return を使う場合は，#include <stdlib.h> は必要ありません．ただし，return を使う場合は，main の返り値は void では間違いです． int main() とします．

演習問題４

4-1 2 から 10000 までの整数の中の素数をすべて表示するプログラムを書きなさい．

4-2 5539 と 24639 の最大公約数を求めるプログラムを書きなさい．

第5章　雑多なこと

これまでに学んだことで様々なプログラムを書くことができます．ここでは落ち葉拾いとして，ここまでに説明していない雑多なことをまとめて説明しておきます．

5.1　型変換

5.1.1　実数型と整数型

`z = x + y;` という文があったとします．x，y，z がすべて整数型で統一されていたり，実数型で統一されていた場合は，何も問題はありません．問題は変数の型が一致しない場合です．

いくつか細かいテクニックはありますが，基本は左辺の代入される側の変数（上の例では `z`）の型と同じ型を右辺で利用します．ただし同じ型を利用できないときもあるので，その場合は，**型変換**するのが簡単です．型変換することを**キャスト**するとも言います．

型変換は以下のように，変数の前にカッコ付きで変換後の型を書くだけです．

（変換後の型）変数

例を示します．

```
int x = 5;
double y;
y = (double)x;
```

上記の例の場合，y には整数型 `x` の値 5 が実数型に変換されて，5.0 となって y に代入されます．もしも上記でキャストがなければ，どうなるのでしょうか？その場合も y の値は 5.0 になります．つまり，自動的にキャストされているのです．整数型を実数型に代入する場合には自動的にキャストされるので，(`double`) は必要ありません．ただし，混乱しないようにつけておくのがよいと思います．例えば，以下の例は誤りやすいはずです．

```
int x = 5;
double y;
y = x/2;
```

この場合，y の値は 2.5 となるのではなく，2.0 になります．なぜなら，まず `x/2` が実行されて，その結果の 2 が型変換されて 2.0 になるからです．2.5 を得るためには，以下のように変数や値をすべて実数型に直すのが確実です．

```
int x = 5;
double y;
y = ((double) x)/2.0;
```

ただしこれは冗長です．実数型と整数型の演算結果は実数型になる ことを利用すれば，以下で十分です．

```
int x = 5;
double y;
y = x/2.0;
```

また変数側を実数型にした以下のような形でもかまいません．

```
int x = 5;
double y;
y = ((double)x)/2;
```

型変換の演算子と割り算の演算子は型変換の演算子の方が順位が高いので，型変換の演算子の方が先に作用します．ですので，上の例では `((double)x)` のカッコは必要ありません．結局，以下のようになります．

```
int x = 5;
double y;
y = (double)x/2;
```

ただし，似たケースとして，以下の場合 2.0 になるので，注意してください．

```
int x = 5;
double y;
y = (double)(x/2);
```

5.1.2 文字型と整数型

プログラムを行う際に文字と数は，区別する必要があります．`'5'` は文字としての 5 であり，`5` は数値のとしての 5 です．

文字`'5'` を数値 5 に変換するには，型変換ではなくて，それ用のプログラムを作る必要があります．単純には以下の文で実現できます．

```
if (x == '0') return 0;
if (x == '1') return 1;
if (x == '2') return 2;
...
if (x == '9') return 9;
```

しかし上記のようなことは実際には行いません．文字と数値の変換には有名なテクニックがあるからです．文字を数値に変換するには，以下の式を使います．

```
x - '0'
```

これは文字のコードが'0'から順に並んでいることと，整数型と文字型の演算は整数型になることを利用しています．数値を文字に直すには，この逆です．

```
(char)(x + '0')
```

とするだけです．ただし x が 0 から 9 までの数値であることはどこかでチェックしていないといけません．

5.2 C 言語は関数型言語

プログラミング言語はいくつかの観点から特徴つけられますが，**関数型言語**であるというのも C 言語の特徴です．

関数型言語の詳しい説明はここでは述べません．ポイントとして，文は必ずある値を返す ということを知っていれば良いと思います．

例えば，式で表現された文はその式の値が**文の返り値**になるし，代入文で表現された文は代入された値が文の返り値になります．

```
1 + 2;        /*  このような文は意味ないが，書いても OK，
                  この式の値 3 がこの文の返り値となる．*/

x = 1 + 2;    /*  この代入文では代入された値が 3 なので，
                  この文の返り値は 3 となる．  */
```

文が値を返すことを利用すると，以下のような文が可能となります．

```
a = b = c = 1;
```

これは変数，a，b，c に 1 を代入しています．まず `c = 1` が実行され，c に 1 が代入されます．この代入文は代入された値 1 を返すので， `b = ～` の ～ の部分が 1 となり，b に 1 が代入されます．同様にして，a にも 1 が代入されます．

さらに文が値を返すことを利用すると冗長な変数を省くことができます．例えば，以下の例では変数 c，d は冗長です．

```
c = a + b;
d = 2 * c;
printf(" %d + %d = %d\n",a,b,c);
printf(" 2 * (%d + %d) = %d\n",a,b,d);
```

これは以下のように書けます.

```
printf(" %d + %d = %d\n",a,b,a+b);
printf(" 2 * (%d + %d) = %d\n",a,b,2*(a+b));
```

しかし，これでは a + b が２度計算されているのでほんの少し効率が悪いかも知れません．気になる人は a + b を変数化してください．代入式の返り値が代入された値であることを利用すると，以下のようにも書けます．ただしこの書き方は読みやすいものではありません．

```
printf(" %d + %d = %d\n",a,b,c = a+b);
printf(" 2 * (%d + %d) = %d\n",a,b,2*c);
```

5.3 条件式の値は０かそれ以外

前章と関連しますが，条件式もある値を返します．実は条件式が返す値が 0 である場合は，偽と判断され，それ以外は真と判断されます.

そのために無限の繰り返しを表すのに，以下のような while 文が使われます.

```
while(1) {
    〜
}
```

これは条件式の値が１で固定であり，必ず真となるので，無限の繰り返しとなります．もちろん，1 という数字は慣習であって，0 以外であれば何でもかまいません．

これを使って 1 から 100 までの和を求めるプログラムを以下のように書いてみました．ただしこのような書き方は読みやすくありません．

```
int main()
{
   int i, sum;

   i = sum = 0;
   while(sum += (i = i + 1)) if (i == 100) break;
   printf("1から 100 までの和は %d です\n",sum);
}
```

[実行結果]

1から 100 までの和は 5050 です

もう少しまともに書くと，以下のようになります.

```
int main()
{
   int i = 101, sum = 0;

   while(i--) sum += i;
   printf("1から 100 までの和は %d です\n",sum);
}
```

演習問題５

5-1 以下の計算結果を実数型で表示したとき，どんな値になるかを示し，なぜそのような値になるのかを説明しなさい．

```
int i = 1;

(1) i/2
(2) i/2.0
(3) (i + 0.0)/2
(4) (double)i/2
(5) (double)(i/2)
```

5-2 以下のプログラムで，a，b，c，は，どんな値になるかを示し，なぜそのような値になるのかを説明しなさい．

```
int main()
{
   int  a, b, c;
   a = (b = (c = 1) + 1) + 2*c;
   printf("a = %d,b = %d,c = %d\n",a,b,c);
 }
```

5-3 上記のプログラムを以下のように変更することができない理由を述べなさい．

```
int main()
{
   int  a, b, c;
   a = 2*c + (b = (c = 1) + 1);
   printf("a = %d,b = %d,c = %d\n",a,b,c);
 }
```

5-4 無限ループを利用して 2 の n 乗 (n = 1,2,3, …) を次々に表示させなさい．ただし整数型の範囲を超える数（予めわかっていないと仮定する）になったら，止まるようにしなさい．

第6章　ヘッダーファイルとライブラリ

6.1　ヘッダーファイルの指定

プログラムの中ではシステムの方から用意されている関数を使うことができます．例えば，printf というのも関数です．関数を使うときには，その関数の宣言がかかれた**ヘッダーファイル**（**インクルードファイル**とも呼ばれる）をプログラムの先頭に指定しなくてはなりません．

例えば，今まで printf という関数を利用してきましたが，これは stdio.h というファイルで宣言されています．そのため printf を使っているプログラムでは，先頭に，

```
#include <stdio.h>
```

が必要になります．

ヘッダーファイルの指定方法には，以下の 2 種類の方法があります．

```
#include <****.h>   と   #include  "****.h"
```

前者の指定方法はシステムが提供しているヘッダファイルを指定する方法です．システムが提供しているヘッダファイルはある特定のディレクトリに存在します．本書の設定では，この特定のディレクトリは `c:\borland\bcc55\include` です．後者の指定の方法は直接ファイルを指定します（絶対パスあるいは相対パスを使います）．自分でヘッダファイルを作成した場合などは後者を使います．

6.2　ライブラリの指定

ヘッダファイルに宣言された関数の本体は，どこかに存在します．通常はプログラムの中か，ライブラリ（システムが提供している関数）です．プログラム中で利用した関数は，コンパイルのときに**リンク**されます．従って，ライブラリの関数を利用する場合には，コンパイルのときにどのライブラリを利用するかを指定する必要があります．指定の方法はコンパイラによって異なります．このテキストの設定ではシステムが提供しているライブラリの指定は省略できます．

ライブラリ関数の利用例としてランダムな正の整数を生成する関数 rand() を使った例題を示します．

練習 6-1　1000 以下のランダムな整数を 100 個生成し表示しなさい．

rand() という関数は呼び出す度に 0 から RAND_MAX の範囲の数をランダムに生成します．以下の式によって，1000 以下のランダムな整数が得られます．

```
rand() % 1000
```

rand() は stdlib.h で定義されているので，プログラムは以下のようになります．

```
#include <stdio.h>
#include <stdlib.h>  /* rand()  を使うので指定が必要 */

int main()
{
    int n;

    for(n = 1;n <= 100;n++) {
       printf("第 %d 回目の数は %d です\n", n, rand()%1000);
    }
}
```

[実行結果]

```
第 1 回目の数は 383 です
第 2 回目の数は 886 です
第 3 回目の数は 777 です
...
第 98 回目の数は 586 です
第 99 回目の数は 94 です
第 100 回目の数は 539 です
```

実はこのプログラムだと，乱数といいながら何度やっても同じ結果しかでません．rand() には乱数の種というものが必要です．種を設定するのは srand() という関数です．srand() に数値を与えることで，種が設定されます．srand() を用いずに，rand() を利用すると，自動的に srand(1) として種を作ったことになります．種を色々かえるためには，srand() に与える数値を変更する必要があります．

以下のようにプログラムが実行された日時を srand() に与えることがよく行われるテクニックです．

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>   /* 構造体 time_t を使うので必要 */

int main()
{
    int n;
    time_t t;

    t = time(0);
    srand((unsigned int)t);

    for(n = 1;n <= 100;n++) {
       printf("第 %d 回目の数は %d です\n", n, rand()%1000);
    }
}
```

[実行結果]

```
第 1 回目の数は 403 です
第 2 回目の数は 297 です
第 3 回目の数は 392 です
...
第 98 回目の数は 937 です
第 99 回目の数は 520 です
第 100 回目の数は 472 です
```

本書では rand() を使う際には，srand() を併せて使うことは行っていません．これは練習だからです．本来なら，上記のようにキチンと書くべきです．

次に sin 関数を使ってみましょう．このような数学関係の関数は math.h で定義されています．また引数に与える実数の単位はラジアンになっています．

練習 6-2 sin(x) の値を x を 0.0 から 7.0 まで 0.1 刻みで計算し表示しなさい．

```
#include <stdio.h>
#include <math.h>  /* sin() を使うので指定が必要 */

int main()
{
    double x;

    for(x = 0.0; x < 7.0;x += 0.1) {
       printf("sin(%f) = %f\n",x,sin(x));
    }
}
```

[実行結果]

```
sin(0.000000) = 0.000000
sin(0.100000) = 0.099833
sin(0.200000) = 0.198669
...
sin(6.800000) = 0.494113
sin(6.900000) = 0.578440
sin(7.000000) = 0.656987
```

最後に,「自分のシステムにはどのような関数が提供されており，それをどう使えばいいのかは，調べないと分からない」ということを注意しておきます．ただし C 言語には標準ライブラリというものが規定されてあるので，基本的なものは書籍で調べることができます．

演習問題６

6-1 stdlib.h や math.h で定義される数学関係のライブラリを調べて，以下の値を実数形式で表示するプログラムを書きなさい．

(1) -2 の絶対値 ===> abs()　返り値の型に注意
(2) 51 の平方根 ===> sqrt()
(3) 底が 10 の log(123.5) ===> log10()
(4) 底が e の log(123.5) ===> log()
(5) e の 2.12 乗 ===> exp()
(6) 1.5 の 3.14 乗 ===> pow()

6-2 ctype.h で定義されている isalpha()，isdigit()，isupper()，islower() の仕様を調べ，簡単な使用例となるプログラムを書きなさい．

第7章　１次元配列

7.1　１次元配列の宣言，代入，参照

配列とは同一の型のデータを複数個まとめて取り扱いたいときに利用するデータ型です.

例） `int ary[5]`

上記の例は int 型のデータが５つあり，それをまとめて取り扱うために ary という名前の配列を宣言しています．例えば， `1,3,10,3,2` という５つのデータをこの配列に順に代入するには，以下のようにします．配列の要素番号は０から始まることに注意して下さい.

```
ary[0] = 1;
ary[1] = 3;
ary[2] = 10;
ary[3] = 3;
ary[4] = 2;
```

また，型の宣言の部分で，以下のように初期化することもできます.

```
int ary[5] = {1,3,10,3,2};
```

ANSI 対応のコンパイラなら，以下のように初期化することもできます.

```
int ary[] = {1,3,10,3,2};
```

このように宣言すると，要素数が 5 であることも同時に宣言したことになります．ただし，これらは初期化であり型の宣言時にだけできる処理です．プログラムの本体内ではこのような形の代入はできません．

配列の特徴としては，配列の添字によって任意の場所を指定できることと，その添字に変数や式を使うことができることです.

練習 7-1 1000 以下の整数をランダムに 100 個生成し，それらを配列 ary に代入しなさい．そして ary の中から最小の値は何番目の要素で最小値は何であるかを表示しなさい.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n,ary[100],minpos,min;

   /*  1000 以下の整数をランダムに 100 個生成 */

   for(n = 0; n < 100; n++) ary[n] =  rand()%1000;

  /*  以下が本体 */

    minpos = 0; min = ary[0];
    for(n = 1; n < 100; n++) {
       if (ary[n] < min) {
          minpos = n; min = ary[n];
       }
    }
    printf("最小値は ary[%d] = %d です\n",minpos,min);
}
```

[実行結果]

最小値は ary[38] = 11 です

7.2 1 次元配列の応用（ソート）

1 次元配列の応用として整数値のソートを行ってみます.

ある数値データを大きい順あるいは小さい順に並べることを**ソート**といいます. ソートには様々なアルゴリズムが存在し, アルゴリズムの違いによってその計算時間は大きく異なります.

ここでは単純なソート法である**選択ソート**というアルゴリズムを使って, 1 次元配列の要素をソートするプログラムを書いてみます.

例えば, 5,3,2,6,4,1 という 6 個の整数を小さい順にソートすることにします. 選択ソートでは, まず 1 番目の要素から最後の要素まで順に見てゆき, 最も小さな値を見つけ, それを 1 番目の要素と取り替えます.

```
{ <5> ,3,2,6,4, <1> } ==> { 1,3,2,6,4,5 }
```

次に 2 番目の要素から最後の要素まで順に見てゆき, 最も小さな値を見つけ, それを 2 番目の要素と取り替えます.

```
{ 1, <3> , <2> ,6,4,5 } ==> { 1,2,3,6,4,5 }
```

これを最後の要素番号まで繰り返せば小さい順に並べ替えることができます.

練習 7-2 1000 以下の整数をランダムに 100 個生成し，それらを配列 ary に代入する．ary の中身を選択ソートにより小さい順に並べ替えて表示しなさい.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n,ary[100],minpos,min,pos,tmp;

    for(n = 0; n < 100;n++) ary[n] =  rand()%1000;

    for(pos = 0;pos < 100; pos++) {

       minpos = pos; min = ary[pos];
       for(n = pos; n < 100; n++) {
           if (ary[n] < min) {
                minpos = n; min = ary[n];
           }
       }

       tmp = ary[pos];      /* ary[minpos] と ary[pos] の交換  */
       ary[pos] = min;      /* いったん tmp を利用していることに注意 */
       ary[minpos] = tmp;
    }

    /*  以下は表示 */

    for(n = 0;n < 100;n++)  printf("ary[%d] = %d\n",n,ary[n]);
}
```

[実行結果]

```
ary[0] = 11
ary[1] = 12
ary[2] = 22
...
ary[97] = 956
ary[98] = 980
ary[99] = 996
```

演習問題７

7-1 1234 を 2 進数に直すプログラムを書きなさい．

7-2 a,b,c,d,e,f,g,h の 8 個の文字から 4 つを選びだす組合わせをすべて表示するプログラムを書きなさい．

7-3 1000 以下の整数をランダムに 100 個生成し，それらを配列 ary に代入します．ary の中身をバブルソートにより小さい順に並べ替えて表示しなさい．

バブルソートはソートのアルゴリズムの１つです．まず配列を最初から見ていき，隣どうしを比較し，逆順ならば交換します．配列の最後までこれを行ったら，また最初に戻って同じことを繰り返します．交換することがなくなるまでこれを続けます．

第8章　関数

8.1　関数の分類

関数は引数，返り値の観点から以下のように４つに分類できます．

$$\begin{cases} \text{引数がない} \begin{cases} \text{返り値が必要} \cdots case1 \\ \text{返り値が不必要} \cdots case2 \end{cases} \\ \text{引数がある} \begin{cases} \text{返り値が必要} \cdots case3 \\ \text{返り値が不必要} \cdots case4 \end{cases} \end{cases}$$

引数とは，通常の関数の変数のことです．$f(x)$ の x をプログラムでは引数と呼びます．**返り値**とは，その関数が返す値（データ）のことです．通常，関数は何らかの処理を行い．その処理結果として，何らかのデータを作成し，その関数を呼び出した部分に，その結果を提供します．このときに提供される値が返り値です．関数が何らかの処理を行い，その処理を行ったということだけが目的である場合，返り値は不必要になります．

8.2　関数の形

一般に関数は以下の形をしています．

```
返り値の型　関数名 （引数の型　引数名，引数の型　引数名,...，引数の型　引数名）
{
    関数の本体
}
```

例えば，以下の関数は返り値が int 型で，関数名は tashizan で引数は int 型の a と int 型の b の２つです．引数 a , b の和である `a+b` の値を返します．本書の分類で言えば case3 （引数あり，返り値が必要）です．

```
int  tashizan (int a, int b)
{
    return  a+b;
}
```

返り値が必要な場合は，返す値を return で返すことに注意して下さい．return 文は以下の形をしています．

```
return 返す値;
```

8.3 関数のプロトタイプ宣言

一般に関数も変数と同じように宣言が必要です．関数の宣言を関数の**プロトタイプ宣言**と言います．

関数の宣言は関数の定義ができていれば，簡単にできます．例えば，前述の関数 tashizan であれば以下です．

```
int  tashizan (int, int);
```

実際の変数名である，a や b を消した形にすればよいのです（ただし配列などは注意が必要です）．

練習 8-1 上記の tashizan 関数を利用して，1 から 100 までの和を計算しなさい．

```
#include <stdio.h>

int tashizan (int, int);  /* 関数のプロトタイプ宣言 */

int main()
{
    int a,sum;

    sum = 0;
    for(a = 1; a < 101; a++) sum = tashizan(sum,a);
    printf("1 から 100 までの和は %d\n",sum);
}

int  tashizan (int a, int b)
{
    return (a+b);
}
```

[実行結果]

```
1 から 100 までの和は 5050
```

上記のプログラムでは，main の中で変数 a が使われています．そして tashizan 関数の中でも変数 a が使われています．通常，変数は**局所変数**となっており，その変数の有効範囲は関数内だけです．つまり，main の中での変数 a と tashizan 関数の中での変数 a は全くの別物です．もしも tashizan 関数の中で main 関数の中の変数は参照したければ，引数にして受け渡すのが普通です．同様に main 関数の中から tashizan 関数の中の変数を参照することもできません．参照したければ，return によって返り値としてわたしてもらうのが普通です．

また本書では関数を main 関数の後に記述しています．同じファイルの中で main 関数の前に関数を記述する場合，関数のプロトタイプ宣言は必要ありません．ただし， main 関数は最初に書き，関数のプロトタイプ宣言も全部行う方が分りやすいと思います．

関数の本体がもう少し複雑なものも作ってみましょう．

練習 8-2 `souwa(x)` を 1 から x までの和と定義します． `souwa(1),souwa(2),souwa(3),…,souwa(100)` を求めなさい．

```
#include <stdio.h>

int  souwa(int);

int main()
{
    int x,sum;

    for(x = 1;  x < 101; x++) {
      sum =  souwa(x);                          /* (1) */
      printf("souwa(%d) = %d\n", x, sum);    /* (2) */
    }
}

int  souwa(int x)
{
   int i,sum = 0;

   for(i = 1; i <= x; i++) sum += i;
   return sum;
}
```

[実行結果]

```
souwa(1) = 1
souwa(2) = 3
souwa(3) = 6
...
souwa(98) = 4851
souwa(99) = 4950
souwa(100) = 5050
```

(1) と (2) の部分は，まとめて以下ように書くのが普通ですが，ここでは返り値があるということを意識させるために，わざと上記の形にしています.

```
printf("f(%d) = %d\n", x, souwa(x));
```

また上記の souwa という関数は以下のようにも書けます.

```
int  souwa(int x)
{
   int sum = 0;
   for(    ; x > 0; x--) sum += x;
   return sum;
}
```

この形だと，変数が一つ減ります．この souwa の中の変数 x と main 関数内の変数 x とが無関係なので，このようなことができることに注意してください.

8.4 引数のない関数

引数のない関数は，呼び出すときには引数の部分に何も書かない形で使います．以前，rand() という関数を使いましたが，これは引数がない関数です.

ただし，引数がない関数の中身を記述する場合には， void という型の宣言が必要です．つまり，もしも rand() の関数を自分で書く場合には，以下のような形になるはずです.

```
int rand(void)
{
    ＊＊＊＊＊＊＊＊＊＊
}
```

8.5 返り値が必要ない関数

返り値が必要ない関数は，呼び出すときには今までの形でかまいません．

ただし，返り値が必要ない関数を記述する場合には， void という型の宣言が必要です．例えば，これまで，printf() という関数を使ってきましたが，これは返り値が必要ない関数である．もしも printf() の関数を自分で書く場合には，以下のような形になると思われます[1]．

```
void printf( ＊＊＊＊＊ )
{
    ＊＊＊＊＊＊＊＊＊＊
}
```

注意として，これまで「返り値が必要ない関数」と言ってきており,「返り値がない関数」とは言っていません．関数の返り値は return 文を使わなくても必ずあります．return 文を使うと何を返すか明示できるので使うのです．

練習 8-3 nprint(x,s) は x 回，文字列 s を表示する関数とする．この関数を作成しなさい．

```
#include <stdio.h>

void  nprint(int,char*);

int main()
{
    nprint(5,"茨城大学"); /* ためしに５回大学名を表示してみよう */
}

void  nprint(int x,char* s)  /* 表示するだけだから返り値は必要ない */
{
   for(   ; x > 0; x--) printf("%s\n",s);
}
```

[1]本当の printf() では返り値は int 型で定義されているはず．

[実行結果]

茨城大学
茨城大学
茨城大学
茨城大学
茨城大学

8.6 再帰呼び出し

C 言語では関数の再帰呼び出しが可能です．再帰呼び出しとは関数の実際の処理の中で，その関数自身を利用することです．例えば funct という名前の関数の本体中で関数 funct を利用することです，

例えば，再び，練習 8-2 の問題を考えてみよう．

練習 8-2（再） souwa(x) を 1 から x までの和と定義します． souwa(1),souwa(2),souwa(3), …,souwa(100) を求めなさい．

この場合， souwa(x) は x = 1 の時は 1 を返し，それ以外の x の時は x + souwa(x-1) という 値を返すことで実現できます．これを利用すると，関数 souwa は以下のようにプログラムするとこができます．

```
int  souwa(int x)
{
   if (x == 1) return 1;
   return x + souwa(x-1);   /* souwa という自分自身の関数を呼び出している */
}
```

再帰呼び出しを利用すると簡潔にプログラムできる場合があります．
関数の練習を行いましょう．

練習 8-4 10000 以下で最も大きな素数を求めるプログラムを書きなさい．

この問題を再びやってみます．このプログラムは，以下の形になっています．

```
int main()
{
    int i;

    for(i = 10000; i > 1; i--) {

       ここに i が素数かどうかを判定し，素数なら表示して
       プログラムを終了する処理を書けば良い

    }
    printf("素数はありませんでした\n");
}
```

i が素数かどうかを判定する関数 primep を作れば良いことがわかります．primep は引数として i を取り，i が素数なら 1 を返し，素数でなければ 0 を返すとします．

結局，main プログラムは，以下のようになります．

```
#include <stdio.h>
#include <stdlib.h>  /* exit を使っているの必要 */

int primep(int);

int main()
{
    int i;

    for(i = 10000; i > 1; i--) {
       if (primep(i)) {
          printf("最大の素数は %d です\n",i);
          exit(0);
       }
    }
    printf("素数はありませんでした\n");
}
```

このプログラムの下に以下の primep の定義を書きます．

```
int primep(int i)
{
    int j;

    for(j = 2; j < i; j++)
      if (i % j == 0) return 0; /* 割りきれる数があったので素数ではない */
    return 1; /* 全部わりきれなかったから素数 */
}
```

[実行結果]

最大の素数は 9973 です

8.7 main も関数

C 言語では利用する関数をすべて宣言する必要があります．またシステムが用意した関数などは対応するヘッダファイルをインクルードすることで宣言できます．関数のプロトタイプは省略できる場合も多いのですが，きちんと，全部宣言すべきです．

そして，実は，main も関数の１つです．なので，きちんと書く場合は main の型も宣言する必要があります．だから，きちんと書くと今までのプログラムは以下のような感じになります．

```
#include <stdio.h>

int main(void);     /* ここまで書く必要はない */

int main(void)      /* この書き方も少しくどい */
{
  ....
}
```

もちろん，main 関数が引数を持ったり，返る値が必要であったりする場合には，void の部分は相応のものを書く必要があります．ただし，main のプロトタイプ宣言まで書くのは冗長です．書く必要はありません．

8.8 call by value

プログラム言語の関数の引数の実装方法には call by reference (**参照呼び出し**) と call by value (**値呼び出し**) という 2 種類の方法があります．call by reference の場合，引数は引数を参照してい

るポインタが渡されます．call by value の場合，引数にはその値が渡されます．値が渡された場合，その値は関数内でコピーされます．そのため呼び出した側の引数と関数に渡された引数は渡された段階で値は同じですが，全く別の変数です．C 言語は call by value です．

練習 8-5 以下の例で call by value を確認しなさい．(2) の部分で x は幾つになっているでしょうか？

```
#include <stdio.h>

int f(int);

int main()
{
   int x,y;

   x = 1;
   y = f(x);                    /* (1) */
   printf("x は %d です\n",x); /* (2) */
   printf("y は %d です\n",y);
}

int f(int x)    /* (3) */
{
   x = x + 2;   /* (4) */
   return x;
}
```

[実行結果]

```
x は 1 です
y は 3 です
```

答は `x = 1` でした．(1) で関数 f に x を渡した段階で x は 1 です．この 1 が (3) の x にコピーされます．この (3) の変数 x は (1) の x とは別の変数です．(4) では関数 f 内の x に 2 が足され，それが x に代入されます．そして関数の処理が終わり，x の値を返して main の (1) に処理が戻ります．そして y に 3 が代入されます．大事なのは，main の中の x の値は関数を呼び出したときの値 1 から変化していないことです．

まとめておくと，関数 f 内の x と main の中で関数 f の引数になった x は全く別物です．関数 f が呼び出されたときに，関数 f 内の局所的な変数 x に引数の値である 1 がコピーされるだけです．

演習問題８

8-1 a の b 乗を計算する関数 power(a,b) を作成しなさい．ただし，a, b は正の整数とします.

8-2 a の b 乗を計算する関数 power(a,b) を作成しなさい．ただし，a, b は実数とします.（ヒント，a の b 乗は log や e で表すとどうなるかを考えてみなさい.）

8-3 i 番目に小さい素数を求める関数 nthprime(i) を作成しなさい．例えば，2 が 1 番小さい素数，3 が 2 番目に小さい素数なので，`nthprime(1) = 2`，`nthprime(2) = 3` です.

8-4 a の階乗を計算する関数を再帰を用いて書きなさい.

第9章　ポインタ

9.1　ポインタはアドレスを値とする変数

C 言語は call by value であることは述べました．しかし与えた引数を関数を使って変更したい場合もあります．

例えば，変数 a と 変数 b の値を交換したい場合以下のようにします．

```
int main()
{
   int a,b,tmp;

   a = 2; b = 5;
   printf("a = %d , b = %d\n",a,b);
   tmp = a; a = b; b = tmp;          /* a と b を交換 */
   printf("a = %d , b = %d\n",a,b);  /* 表示して確認 */
}
```

[実行結果]

```
a = 2 , b = 5
a = 5 , b = 2
```

変数 tmp を使いたくないので，変数 a と 変数 b を交換する関数 swap を作ってみましょう．単純に swap(a,b) では実現できません．call by value なので関数から戻ったら，a や b の値はもとのままだからです．工夫が必要です．

ここで**ポインタ**の登場です．変数 x は箱です．その変数に `&` をつけた変数はその変数（箱）の場所を示す**アドレス**です．このアドレスを示す変数のことをポインタといいます．つまりポインタはアドレスを値とする変数です．またアドレス自身は整数と考えて構いません．

`x = 15;`　　としてみます．

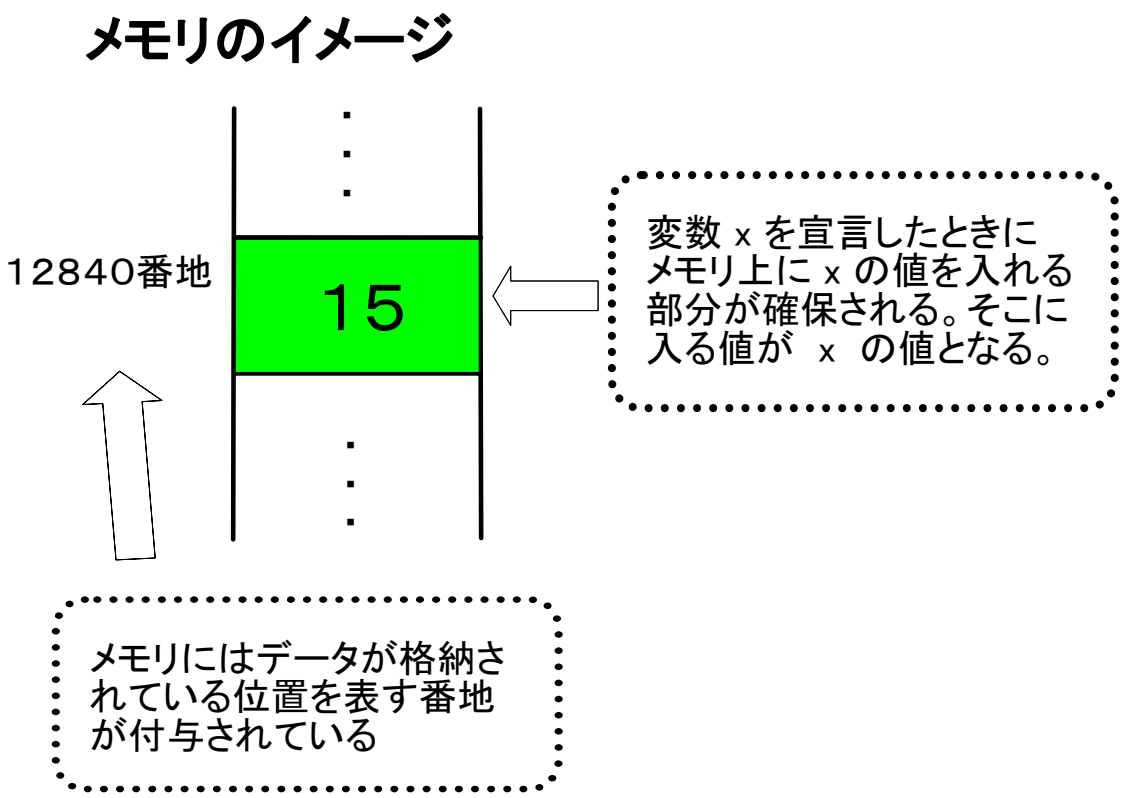

図 9.1: メモリ上の変数

これは 変数 x の中身，つまり x という名の箱の中に 15 を入れる操作です．この箱はメモリ上のどこかに存在しているので，メモリ上でその箱 x の位置を示すアドレスが存在します．そのアドレスは **& x** で参照できます．つまり変数 x のポインタは ＆x となります．またポインタの値であるアドレスに存在する箱の中身はポインタの変数に * をつけることで参照できます．

つまり以下の関係があります．

```
int  x      <=== 整数型の変数
    &x      <=== 上記の変数の実体があるメモリ上のアドレス
   *(&x)    <=== ポインタ &x の指すアドレスの中身
```

つまり　*(&x) = x　の関係は常に成立

```
int *y      <=== ポインタの宣言
                 この宣言により以下の２つのことを示している．
                 (1) 変数 y はポインタである．
                 (2) 変数 y が示すアドレスに存在する値の型は int 型
```

9.2 関数にポインタを渡す

前述した swap の実現のために，次に関数の引数にポインタを使う方法を説明します．

練習 9-1 以下のプログラムにおいて，funct の呼び出し前後で変数 a の値は変化しているが，a のアドレス&a は変化していないことを確認しなさい．

```
#include <stdio.h>

void funct(int *);  /* このプロトタイプの書き方に注意！ */

int main()
{
   int a = 1;

   printf("a = %d, &a = %d\n",a,&a);
   funct(&a);
   printf("a = %d, &a = %d\n",a,&a);
}

void funct(int *x)  /* この書き方は特に注意！ */
{
   *x += 2;
}
```

[実行結果]

```
a = 1, &a = 2147482180
a = 3, &a = 2147482180
```

まず関数の引数の宣言部分に注意して下さい．引数には型の指定が必要ですが，引数がポインタのときは，そのポインタの指すアドレスの中身がどのような型になるかを指定しなくてはいけません．上記の場合，関数 funct の引数は整数型の変数 x へのポインタです．関数の呼び出しは簡単です．引数にはポインタを渡せばよいだけです．問題は関数の引数の定義の部分です．以下のように書くことで，引数は int 型の変数を指すポインタだと宣言したことになります．

```
int *x
```

練習 9-1 のプログラムでは funct に渡されるのは変数 a のアドレスです．call by value なので，funct から戻っても引数として利用した値（a のアドレス）は変化しません．ただし，アドレスが指している中身（つまり a の値）は変化しています．

以上のことを利用すれば， swap 関数は以下のように実現できます．

```
#include <stdio.h>

void swap(int *,int *);

int main()
{
   int a,b;

   a = 2; b = 5;
   printf("a = %d , b = %d\n");
   swap(&a,&b);
   printf("a = %d , b = %d\n");
}

void swap(int *a,int *b)
{
   int tmp;
   tmp = *a; *a = *b; *b = tmp;
}
```

[実行結果]

```
a = 5 , b = 2
a = 2 , b = 5
```

9.3 配列とポインタの関係

配列とポインタには，以下の非常に重要な関係があります．

> 配列の最初の要素（ary[0]）が入っているメモリ上のアドレスは配列の名前（ary）で参照できます．つまり配列へのポインタは配列名です．

これは以下のプログラムでも確認できます．ただし，このプログラムではアドレスの値を整数型として扱っていますが，これは本当はやってはいけません．

```
#include <stdio.h>

 int main()
 {
    int ary[5] = {10,20,30,40,50};
    printf("ary[0] のアドレスは %d\n",&(ary[0]));
    printf("ary  の値は          %d\n",ary);
 }
```

[実行結果]

```
          ary[0] のアドレスは 125048
          ary  の値は          125048
```

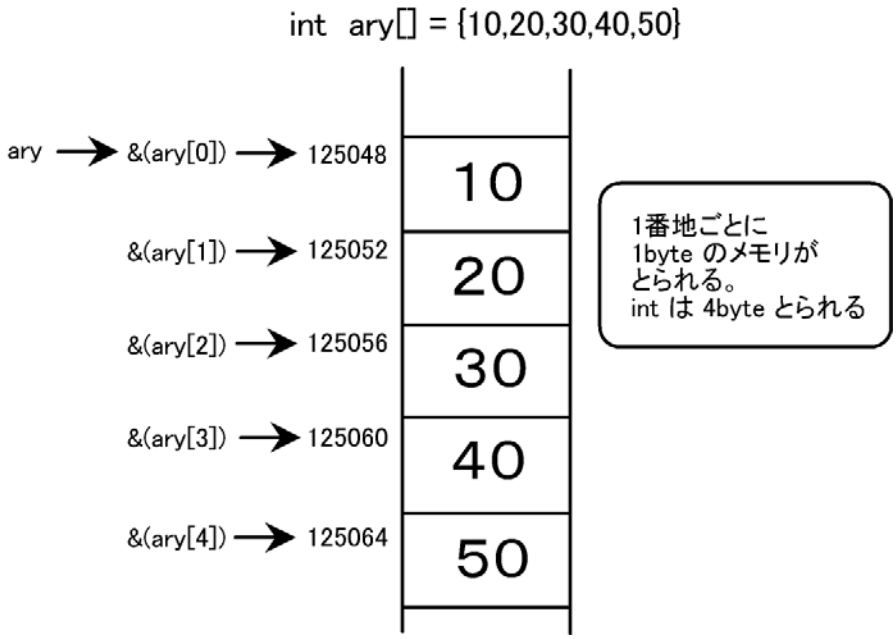

図 9.2: メモリ上の配列の様子

また以下の関係もあります.

$$\mathrm{ary}[0] = {}^*\mathrm{ary}$$

これは以下のプログラムで確認できます.

```
#include <stdio.h>

 int main()
 {
    int ary[5] = {10,20,30,40,50};
    printf("ary[0] = %d\n",ary[0]);
    printf("*ary   = %d\n",*ary);
 }
```

[実行結果]

```
          ary[0] = 10
          *ary   = 10
```

配列の 2 番目の要素（ary[1]）のアドレスは ary + 1 で参照できます．つまり以下の関係式も成立します．

$$\mathrm{ary[i]} = {}^{*}(\mathrm{ary} + \mathrm{i})$$

これは以下のプログラムで確認できます．

```
 int main()
 {
    int ary[5] = {10,20,30,40,50};
    int i;

    for(i = 0;i < 5;i++)
        printf("ary[%d] = %d\n",i,ary[i]);

    for(i = 0;i < 5;i++)
        printf("*(ary + %d) = %d\n",i,*(ary + i));
 }
```

[実行結果]

```
          ary[0] = 10
          ary[1] = 20
          ary[2] = 30
          ary[3] = 40
          ary[4] = 50
```

```
*(ary + 0) = 10
*(ary + 1) = 20
*(ary + 2) = 30
*(ary + 3) = 40
*(ary + 4) = 50
```

9.4 配列を関数に渡す

配列を関数に渡すには配列のポインタ，つまり先頭の要素が入っているアドレスを渡します．そして先頭の要素が入っているアドレスは，前章で説明した通り配列名です．あとは整数のポインタと考え方は同じです．

関数内で配列の中身を変更した場合，関数から main プログラムに処理が戻ると配列の中身は変更されていることに注意してください．

練習 9-2 配列 ary の要素を逆順にする関数 reverse を作りなさい.

問題は関数 reverse の引数の定義の部分です. 以下の答で配列を引数とする場合の書き方を学んでください.

```
#include <stdio.h>

void reverse(int []);  /* この書き方は注意 !!! */

int main()
{
   int ary[10] = {0,1,2,3,4,5,6,7,8,9};
   int i;

   for(i = 0;i < 10;i++) printf("ary[%d] = %d\n",i,ary[i]);
   reverse(ary);
   for(i = 0;i < 10;i++) printf("ary[%d] = %d\n",i,ary[i]);
}

void reverse(int ary[])  /*  この書き方も注意！  *ary でもよい */
{
   int ary2[10],i;

   for(i = 0;i < 10;i++) ary2[i] = ary[i];
   for(i = 0;i < 10;i++) ary[i] = ary2[9 - i];
}
```

[実行結果]

```
ary[0] = 0
ary[1] = 1
ary[2] = 2
...
ary[7] = 7
ary[8] = 8
ary[9] = 9
ary[0] = 9
ary[1] = 8
ary[2] = 7
ary[3] = 6
...
ary[7] = 2
```

```
ary[8] = 1
ary[9] = 0
```

同じような例題を 2 つ示します.

練習 9-3 配列 ary の要素の中で最大値を与える添字を求める関数 getmaxpos を作りなさい．つまり，ary[i] が最大だとしたら，i を返す関数です.

```
#include <stdio.h>

int getmaxpos(int []);

int main()
{
   int ary[10] = {0,1,2,3,4,5,6,7,8,9};
   int maxpos;

   maxpos = getmaxpos(ary);
   printf("最大値は ary[%d] = %d です\n",maxpos,ary[maxpos]);
}

int getmaxpos(int ary[])
{
   int i,maxpos;

   maxpos = 0;
   for(i = 1;i < 10;i++)  if (ary[i] > ary[maxpos]) maxpos = i;
   return maxpos;
}
```

[実行結果]

```
最大値は ary[9] = 9 です
```

練習 9-4 配列 ary の要素の中で，しかも添字が i から j の範囲の中で最大値を与える添字を求める関数 getmaxpos を作りなさい.

```
#include <stdio.h>

int getmaxpos(int [],int,int);

int main()
{
   int ary[10] = {0,1,2,3,4,5,6,7,8,9};
   int maxpos,i,j;

   i = 2; j = 7;
   maxpos = getmaxpos(ary,i,j);
   printf("%d と %d の間で最大値は ary[%d] = %d です\n",
                                        i,j,maxpos,ary[maxpos]);
}

int getmaxpos(int ary[],int i,int j)
{
   int maxpos;

   maxpos = i;
   for(     ;i <= j;i++) if (ary[i] > ary[maxpos]) maxpos = i;
   return maxpos;
}
```

[実行結果]

```
2 と 7 の間で最大値は ary[7] = 7 です
```

練習 9-5 ランダムに生成した 1000 以下の 100 個の整数を選択ソート法により大きい順に並べるプログラムを書きなさい.

選択ソートは，以下の順に行われる.

(1) 「0 番目の要素」と「0 番目から 99 番目までの最大値の要素」と交換する.
(2) 「1 番目の要素」と「1 番目から 99 番目までの最大値の要素」と交換する.
(3) 「2 番目の要素」と「2 番目から 99 番目までの最大値の要素」と交換する.

…

(99) 「98 番目の要素」と「98 番目から 99 番目までの最大値の要素」と交換する.

```
#include <stdio.h>
#include <stdlib.h>

int getmaxpos(int [],int,int);

void select_sort(int []);

int main()
{
   int i,ary[100];

   for(i = 0;i < 100;i++) ary[i] = rand()%1000;
   select_sort(ary);

   for(i = 0;i < 100;i++)
     printf("%i 番目に大きい数は %d です\n",i,ary[i]);
}

void select_sort(int ary[])
{
   int i,j,tmp;

   for(i = 0;i < 99;i++) {
      j = getmaxpos(ary,i,99);
      tmp = ary[i]; ary[i] = ary[j]; ary[j] = tmp;
   }
}

int getmaxpos(int ary[],int i,int j)
{
   ここはすでに書いた
}
```

[実行結果]

```
0 番目に大きい数は 996 です
1 番目に大きい数は 980 です
2 番目に大きい数は 956 です
...
97 番目に大きい数は 22 です
98 番目に大きい数は 12 です
99 番目に大きい数は 11 です
```

演習問題９

9-1 a，b，c にはそれぞれ数値が入っているとします．その 3 つの数値を大きい順で a，b，c に入れ直すプログラムをかきなさい.（ヒント，swap をつかうと簡単）

9-2 配列に入っている数値データの平均を求める関数を作りなさい．ただし，配列を引数とすること．

9-3 配列に入っている数値データの分散を求める関数を作りなさい．ただし，配列を引数とすること．

9-4 正の整数 i を 8 進数で表示するとき，8^n の位の数値がいくつになるかを調べる関数 npos8(i,n) を作成しなさい． `n = 0,1,2` … ．

例えば，80 は 8 進数で表すと 120 なので，npos8(80,0) = 0，npos8(80,1) = 2，npos8(80,2) = 1 です.

第10章　文字列

ここでは，日本語の文字列は考えずに，半角英数字だけを扱うことを考えます．日本語の文字列を扱うにはコードの問題があるので少し面倒です．

10.1　文字列は文字の配列

文字列というデータ型はC言語にはありません．char 型の配列として実現します．例えば，”Ibaraki” という文字列をデータとして実現するには，基本的には以下のように文字の配列を用意し，各要素に文字を代入することで実現します．

```
char name[8];

name[0] = 'I'; name[1] = 'b'; name[2] = 'a'; name[3] = 'r';
name[4] = 'a'; name[5] = 'k'; name[6] = 'i'; name[7] = '\0';
```

文字列を表す配列には文字列の最後を示す '\0' という記号が入ります．このため長さ x 文字の文字列を表現するには大きさ x + 1 の配列が必要です．

10.2　文字列の表示

基本的には配列の各要素である char 型のデータを一つづつ順に出力させれば表示できます．

```
int main()
{
    char name[100];  /* 大きさは適当に大きければ良い */
    int i;

    name[0] = 'I'; name[1] = 'b'; name[2] = 'a'; name[3] = 'r';
    name[4] = 'a'; name[5] = 'k'; name[6] = 'i'; name[7] = '\0';

    for(i = 0;name[i] != '\0';i++)  printf("%c",name[i]);
                                          /* 文字の出力は %C */
    printf("\n");
}
```

[実行結果]

```
        Ibaraki
```

printf の中のフォーマットで　%s　を使う方法もあります．こっちの方が簡単です．

```
int main()
{
    char name[100];

    name[0] = 'I'; name[1] = 'b'; name[2] = 'a'; name[3] = 'r';
    name[4] = 'a'; name[5] = 'k'; name[6] = 'i'; name[7] = '\0';

    printf("%s\n",name);   /* 変数にあたる部分は配列名 */
}
```

[実行結果]

```
        Ibaraki
```

10.3　文字列への代入

基本的には先ほど示したように，配列の各要素に各文字を代入することによって文字列への代入ができます．

配列と同じように初期化するさいにまとめて代入することもできます.

```
char name[] = {'I','b','a','r','a','k','i','\0'};
```

さらに，文字列の場合は，以下のような簡単な形で初期化できます.

```
char name[] = "Ibaraki";
```

同じような書き方として，以下の書き方もできます.

```
char *name = "Ibaraki";
```

ただしこれらは初期化だけで通用する方法です．プログラムの本体内でこの書き方はできません．
また下のプログラムの場合，name の配列の大きさは「文字の数＋１」（上記の場合８）で固定されます．そのため文字列の長さが長くなるようなら，この方法で代入してはいけません．

```
int main()
{
    char name[] = "Ibaraki";
    printf("%s\n",name);
}
```

[実行結果]

```
Ibaraki
```

文字列への代入は通常，ライブラリ strcpy を使います．以下の形で，文字列配列２の中身が文字列配列１へコピーされます.

```
strcpy(文字列配列１のポインタ，文字列配列２のポインタ)
```

文字列配列のポインタとは配列名です.
また２重引用符で囲まれた文字列自身も文字列配列のポインタです.

```
#include <stdio.h>
#include <string.h>    /* strcpy を使う場合のヘッダーファイル */

int main()
{
    char name1[100],name2[100];

    strcpy(name1,"Ibaraki");
    printf("%s\n",name1);

    strcpy(name2,"University");
    printf("%s\n",name2);

    strcpy(name1,name2);  /* name2 を name1 にコピー */
    printf("%s\n",name1);
}
```

[実行結果]

```
Ibaraki
University
University
```

練習 10-1 文字列を反転させる関数 reverse を書きなさい.

```
#include <stdio.h>
#include <string.h>

void reverse(char []);

int main()
{
    char univ[] = "Ibaraki University";

    printf("%s\n",univ);
    reverse(univ);       /* univ の中身は変更される */
    printf("%s\n",univ);
}

void reverse(char ary[])
{
   char ary2[100];  /* 適当に大きめに取る */
   int i,k;

   strcpy(ary2,ary);  /* ary を ary2 にコピー */

   for(i = 0;ary[i] != '\0';i++);   /* 文字列の最後の要素の位置 i を調べる */

   k = 0;
   for(i--; i >=0;i--) {
     ary[k] = ary2[i]; k++;
   }
}
```

[実行結果]

```
          Ibaraki University
          ytisrevinU ikarabI
```

10.4 文字列のその他の操作

文字列に対していろいろな操作を行う関数が string.h で定義されています．ひとまず大事なものは，strcpy 以外では以下の３つです．

● strlen(str) ==> 文字列 str の長さを返す

ex) char str[] = "Ibaraki";
strlen(str) ==> 7

● strcmp(str1,str2) ==> 文字列 str1 と str2 を辞書式で比較する．等しかったら 0
str1 の方が後 (大きかったら) 0 以上の正の値

ex) char str1[] = "Ibaraki";
char str2[] = "Hitachi";
strcmp(str1,str2) ==> str1 の方が後なので，正の値が返る

● strstr(str1,str2) ==> 文字列 str1 から文字列 str2 を検索する．str1 の中に str2 を
発見したら，str1 における発見した位置のポインタを返す．
発見できなかったら NULL を返す．

ex) p = strstr("Ibaraki","ara");
printf(p) ==> "araki"

これらを使うと先の reverse は以下のようにも書けます．

```
void reverse(char ary[])
{
   char ary2[100];  /* 適当に大きめに取る */
   int i,k;

   strcpy(ary2,ary);  /* ary を ary2 にコピー */
   i = strlen(ary);   /* 文字列の最後の要素の位置を調べる */
   k = 0;
   for(i--; i >=0;i--) {
     ary[k] = ary2[i]; k++;
   }
}
```

少し考えれば，ary2 を使わなくてもできることがわかります．

```
void reverse(char ary[])
{
   int k,midpos,len;
   char tmp;

   len = strlen(ary);
   midpos = len/2;
   for(k = 0; k <= midpos;k++) {
     tmp = ary[k];
     ary[k] = ary[len - 1 - k];
     ary[len - 1 - k] = tmp;
   }
}
```

演習問題１０

10-1 文字列の子音だけを取り出して表示するプログラムを作成しなさい.

```
"Hitachi"  --->  "Htch"
```

10-2 姓と名を逆にして表示するプログラムを書きなさい.

```
"Suzuki Ichiro"  --->  "Ichiro Suzuki"
```

10-3 ある文字列が正の整数を表しているかどうか調べる関数 `checki` を作りなさい. `checki` は引数の文字列が正の整数なら 1 を返し，それ以外なら 0 を返すとします.

```
checki("123")  ==> 1
checki("1a2")  ==> 0
```

10-4 文字列 str1 の中に "Ibaraki" と "Hitachi" がともにこの順番で含まれるかどうかを判定する関数を書きなさい.

第11章　２次元配列

11.1　２次元配列は１次元配列の配列

２次元配列は１次元配列の配列です.

例として，クラス名簿を考えてみます．各人の名前は 1 次元配列で表せます．例えば，名前は 30 文字以下として各人の名前は `name[30]` で実現できます． `name[30]` という 1 次元配列を人数分 (40 名分) 用意したい場合には,

```
name[40][30]
```

という具合いに宣言します.

縦横の関係を逆転させた場合は,

```
name[30][40]
```

となります．ただし，これは効率が悪いので，通常は以下のように実現します.

2 次元配列の名前 [1 次元配列の個数] [1 次元配列の大きさ]

11.2　２次元配列は 1 次元配列のポインタの配列

２次元配列 `name[40][30]` を考えます．先の説明の通り，これは大きさ 30 の１次元配列が 40 個あることを示しています.

このとき， `name[i]` のあらわすものは， i 番目の大きさ 30 の１次元配列へのポインタです. つまり 2 次元配列は 1 次元配列のポインタの配列なのです.

大きさ 30 の１次元配列が文字列を表す場合，つまり文字の配列の場合，その文字列の表示を行うには，そのポインタを利用して，以下のように実現できます.

```
printf("%s\n",name[i]);
```

練習 11-1 3 名の名前 Suzuki， Tanaka， Sato を 2 次元配列に入れ，それらの名前を表示しなさい.

```
#include <stdio.h>
#include <string.h>  /* strcpy を使うなら必要 */

void reverse(char *);

int main()
{
    char name[3][30];  /* 3 名分の名前（30 文字以下） */
    int  i;

    name[0][0] = 'S'; name[0][1] = 'u'; name[0][2] = 'z';
    name[0][3] = 'u'; name[0][4] = 'k'; name[0][5] = 'i'; name[0][6] = '\0';

    /*  あるいは  strcpy(name[0],"Suzuki");      */

    name[1][0] = 'T'; name[1][1] = 'a'; name[1][2] = 'n';
    name[1][3] = 'a'; name[1][4] = 'k'; name[1][5] = 'a'; name[1][6] = '\0';

    /*  あるいは  strcpy(name[1],"Tanaka");      */

    name[2][0] = 'S'; name[2][1] = 'a'; name[2][2] = 't';
    name[2][3] = 'o'; name[2][4] = '\0';

    /*  あるいは  strcpy(name[2],"Sato");      */

    for(i = 0;i < 3;i++) printf("%s\n",name[i]);

    /* name[i] は i 番目の 1 次元配列に対するポインタになっている */
}
```

[実行結果]

```
Suzuki
Tanaka
Sato
```

11.3 ２次元配列を関数へ渡す

２次元配列を関数に渡す場合には，ポインタの概念や配列のメモリ内での実装方法を知らないときちんと理解はできません．

ここでは方法だけを示します．このやり方をそのまま覚えて下さい．２次元配列を `char name[40][30]` ，関数を `funct` とします．

呼び出し側 呼び出し側は簡単です．配列名だけで構いません．

```
funct(name)
```

関数の引数の書き方 呼び出される関数の引数の書き方が問題です．以下の形をとります．

```
funct(char name[][30])
```

大事なのは，２つめの括弧内に配列の大きさが必要なことです．

関数の宣言の書き方 これは関数の書き方が分かっていれば問題ありません．

```
void funct(char [][30])
```

void の部分は関数 funct の返り値の型を書きます．ここでは仮に void としています．

重要な点として，１次元配列を関数に受け渡した場合も同じですが，関数内で配列の内容を変更した場合，処理が main に戻っても中身が変更されていることです．これは関数には実際はポインタを渡しており，関数内部ではポインタの指す実際の値そのものを変更しているからです．

練習 11-2 名前を表す文字列の集合である 2 次元配列 name を考えます．name の中の名前を辞書順で並べた場合に，最も先頭にくる名前を表示するプログラムを作成しなさい．

`name[0],name[1],name[2],…` を名前の集合と見なして，辞書式で最も先頭にくる名前の要素番号 i を返す関数を作成し，それを利用すればできます．

```
#include <stdio.h>
#include <string.h>

int dic_first(char [][30]);

int main()
{
    char name[10][30];  /* 10 名以下の名前（30 文字以下） */
    int  fpos;

    strcpy(name[0],"Suzuki");
    strcpy(name[1],"Tanaka");
    strcpy(name[2],"Sato");

    name[3][0] = '\0';  /* データの終りの印をつけておく */

    fpos = dic_first(name);

    printf("最初の名前は  %s  です\n",name[fpos]);
}

int dic_first(char name[][30])
{
    int min,i,cmp;

    min = 0;
    for(i = 1;name[i][0] != '\0';i++) {
       cmp = strcmp(name[i],name[min]);
       if (cmp < 0) min = i;
    }
    return min;
}
```

[実行結果]

最初の名前は Sato です

演習問題１１

11-1 20 人の適当な名前（半角アルファベット）を `name[20][30]` に代入し，それらを辞書順にソートする関数を作成しなさい．そしてそのソート結果を表示するプログラムを作成しなさい．名前の入力はプログラムの中で，20 人分書くこと．

11-2 以下の行列 A を 2 次元配列で表現し，その配列の転置行列 A^t を作る関数を作りなさい．第 1 引数が行列 A，第 2 引数に計算結果の行列 A^t が入るようにしなさい．

$$A = \begin{bmatrix} 1 & -2 & 3 & -1 & 5 \\ 0 & -2 & 1 & 0 & 3 \\ -1 & 0 & 0 & 1 & 3 \\ 2 & -1 & 2 & 4 & 0 \\ -1 & -1 & 1 & 1 & 1 \end{bmatrix}$$

第12章　プログラムの引数

12.1　プログラムへ引数を渡す

記述したプログラムを myprog.c でセーブし，コンパイルすることで実行ファイル myprog.exe を作成し，

```
> myprog     Enter
```

によってプログラムが実行できます．このプログラムに引数を渡したいときがあります．例えば以下のような感じです．

```
> myprog 123 321 Ibaraki     Enter
```

上記の ”123”，”321”，”Ibaraki” が**プログラムの引数**です．この引数をプログラムで利用する方法を説明します．

これは簡単です．main の定義を以下のように書けばできます．

```
int main()    /* 今までの書き方（１） */

int main(int argc, char *argv[])    /* 今後の書き方（２） */
```

これは大事です．きちんとしたプログラムを書くためには，今後（２）の書き方をすべきです．（２）の書き方をした場合，変数 `argc` や `argv` には以下のようなデータが代入されます．

`argc` ==> 「引数の個数＋１」が入る．整数型のデータ．
上記の例では引数が３つあるので `argc = 4` となる．

`argv[0]` ==> プログラム名が文字列として入る．ポインタが入る．
上記の例では `argv[0] = "myprog"`
絶対パスで示される場合もある．

`argv[1]` ==> 第１引数が文字列として入る．ポインタが入る．
上記の例では `argv[1] = "123"`

argv[2] ==> 第 2 引数が文字列として入る．ポインタが入る．
上記の例では argv[2] = "321"

argv[3] ==> 第 3 引数が文字列として入る．ポインタが入る．
上記の例では argv[3] = "Ibaraki"

以下どんどん続く．
argv は文字列を表す 1 次元配列の集合，つまり 2 次元配列だと考えることもできます．

練習 12-1 上記の記述を確かめるプログラムを書きなさい．

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

  printf("プログラムの名前は  %s  です. \n",argv[0]);
  printf("プログラムの引数は  %d  個です. \n",argc - 1);

  for(i = 1;i < argc;i++) {
     printf("第 %d 引数は  %s です. \n",i,argv[i]);
  }
}
```

上記のプログラムをディレクトリ \home の下に myprog.c という名で作り．コンパイルする．そして例えば以下のように実行してみます．

```
> myprog.exe  Ibaraki Hitachi 12 -2.6   Enter
```

この実行結果は以下のようなものです．

```
プログラムの名前は  C:\HOME\MYPROG.EXE  です.
プログラムの引数は  4  個です.
第 1 引数は Ibaraki  です.
第 2 引数は Hitachi  です.
第 3 引数は 12  です.
第 4 引数は -2.6  です.
```

12.2 文字列を整数に変換する（atoi）

練習 12-2 プログラムに整数の引数を 1 つつけて，その数の 3 倍の数を表示するプログラムを書きなさい.

実行例

```
> sanbai 124
124 * 3 = 372
>
```

ポイントは文字列の ”124” をどうやって整数の 124 に変換するかです．これは文字列を整数に変換する関数 atoi を利用すればできます.

```
atoi("124") ---> 124
```

あるいは

```
char suchi[4];
suchi[0] = '1';suchi[1] = '2';suchi[2] = '4';suchi[3] = '\0';
atoi(suchi)  ---> 124;
```

あるいは

```
char str[] = "124";
atoi(str)  ---> 124;
```

```
#include <stdio.h>
#include <stdlib.h>   /* exit や atoi を使うから必要 */

int main(int argc, char *argv[])
{
  int i;

  if (argc != 2) { printf("引数が違う\n"); exit(0); }

  printf("%s * 3 = %d\n",argv[1],3*atoi(argv[1]));
}
```

[実行結果]

```
> prog 15

15 * 3 = 45
```

もう一つ例題をやってみましょう.

練習 12-3 プログラムに２つの整数を引数として，それら引数が表す数値の和を表示するプログラムを書きなさい.

実行例

```
> wa 124 36
124 + 36 = 160
>
```

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  int a,b;

  if (argc != 3) { printf("引数が違う\n"); exit(0); }

  a = atoi(argv[1]);
  b = atoi(argv[2]);
  printf("%s + %s = %d\n",argv[1],argv[2],a+b);
}
```

プログラムの引数は文字列となるので，文字列を数値に変換する atoi はよく使う関数です．以下 atoi について述べていなかった点を補足します.

● `atoi` は負の数も処理できる.

`atoi("-12") --> -12` が返る.

● `atoi` は文字列の最後が `'\0'` ではなくて，`'\n'` であっても同様に処理を行う.

```
char num[] = {'-', '1', '2', '\n'};
atoi(num) --> -12 が返る.
```

● `atof` という関数もある． `atoi` は整数が対象だったが， `atof` は倍精度の実数が対象.

`atof("-12.1") --> -12.1` が `double` 型の実数として返る.

演習問題１２

12-1 プログラムに渡された姓と名を逆にして表示するプログラム revname を書きなさい.

実行例

```
> revname Suzuki Ichiro
Ichiro Suzuki
```

12-2 プログラムに複数個（任意個である点に注意）の整数を引数として渡し，それら引数が表す数値の和を表示するプログラム sum を書きなさい.

実行イメージ

```
> sum 3 4 56 7
70

> sum 35 6
41

> sum 1 2 4 56 7 101
171
```

12-3 プログラムに複数個（任意個である点に注意）の英単語を引数として渡し，その単語の文字列を大文字に変換して表示するプログラム toup をかきなさい.

実行例

```
> toup Ibaraki
IBARAKI

> toup Ibaraki University
IBARAKI UNIVERSITY
```

第13章　ファイルの入出力

ここで扱うファイルはテキストファイルとします．テキストと言っても文書が入っているファイルだけを想像してはいけません．データがテキスト形式で入っている場合もよくあります．テキストファイルを開いてそのファイルに書かれている内容を読んで，ある処理をしたり，また，プログラムの結果をファイルに書き出したりすることを学習します．

13.1　ファイルのオープンとクローズ

ファイルを操作するには，読むにしても，書くにしても，とにかく最初にそのファイルをオープンする必要があります．次に何か処理をして，最後にクローズします．クローズは忘れがちなので，注意して下さい．

ファイルのオープンには fopen という関数を使います．

```
FILE *fp;
fp = fopen(ファイル名，モード);
```

fp は FILE という型へのポインタです．**ファイルポインタ**と呼ばれます．FILE という型は，構造体です．まだ説明していないので，説明は省略します．ここでは詳しく知らなくても構いません．こういうものだと思っていて下さい．オープンしたファイルはファイルポインタを介してアクセスできます．fopen はファイルを開いて，開いたファイルに対するファイルポインタを返します．上記のように，fopen は 2 つの引数をもちます．第 1 引数はファイル名，つまり文字列です．第 2 引数のモードはファイルを開く目的を指定します．**モード**は幾つかありますが，以下の 3 つが重要です．

```
"r"   ==>    読み込み用
"w"   ==>    新規書き込み用
"a"   ==>    追加書き込み用
```

例として，data.txt というファイルを読み込み用で開いてみます．

```
FILE *fp;
fp = fopen("data.txt", "r");
```

もしもファイル名が間違うなどで，ファイルをオープンできなかったら，fopen は NULL を返します.

ファイルを閉じるには fclose という関数を使います.

```
int r;
r = fclose(ファイルポインタ);
```

flose の返り値は整数です．もしも正常にクローズできれば 0 が返り，何らかの異常があれば EOF が返ります.

13.2 ファイルからの文字の読み出し

オープンされたファイルから文字を読み出す関数として fgetc があります．次のように使います.

```
int c;
c = fgetc(ファイルポインタ);
```

ファイルをオープンした際には，必ず，ファイルのある文字を指しているポインタが存在します. fgetc はそのポインタの指している文字を 1 char 分読み出し，その後，そのポインタを次の文字へ進めます.

- ファイルをオープンした直後では，上記のポインタはファイルの最初の文字を指している.
- fgetc が実行されると，上記のポインタは次の文字へ移る.
- ファイルの最後には EOF という特殊な文字が入っている．これを fgetc で読み込んだらそれ以上 fgetc は行えない.

fgetc の返す値は char なので，受け取る変数を char c; で宣言すれば良さそうですが， EOF は int 型である場合もあるので int c; とする方が安全です.

練習 13-1 適当な内容の書かれたファイル data.txt を作成する．このファイルの内容の 1 行目だけを表示するプログラムを作成しなさい.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    FILE *fp;
    int c;

    fp = fopen("data.txt","r");
    if (fp == NULL) { printf("cannot open file\n"); exit(0); }
    c = fgetc(fp);
    while((c != '\n') && (c != EOF)) {
        printf("%c",c);
        c = fgetc(fp);
    }
    printf("\n");
    fclose(fp);
}
```

[実行結果]

```
> type data.txt
私は、
茨城大学の
学生
です。

> prog
私は、
```

練習 13-2 練習 13-1 のファイル名 data.txt をプログラムの引数として渡すように改良しなさい.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    FILE *fp;
    int c;

    if(argc != 2)  { printf("usage: prog filename\n"); exit(0); }
    fp = fopen(argv[1],"r");
    if (fp == NULL) { printf("cannot open file\n"); exit(0); }
    c = fgetc(fp);
    while((c != '\n') && (c != EOF)) {
        printf("%c",c);
        c = fgetc(fp);
    }
    printf("\n");
    fclose(fp);
}
```

[実行結果]

```
> prog data.txt
私は、
```

練習 13-1，13-2 ともに，文字 c を出力するのに `printf("%c",c)` を使っていますが，文字を出力するだけなら，`putchar(c)` の方が簡単です．

13.3 ファイルからの１行の読み出し (fgets)

オープンしたファイルから１文字づつ読み出す関数 fgetc は先ほど説明しました．C 言語には１行づつ読み出すという便利な関数 `fgets` があります．次のようにして使います．

```
char *c;
c = fgets(文字型の配列名，その配列の大きさ，ファイルポインタ);
```

fgets はファイル中の文字の位置を示すポインタを返します．もしも最後まで読み込んでいて，これ以上読み込めない場合は NULL を返します．通常，返る値はファイルの終りを検出する以外は使われません．読み込んだ１行が第１引数の配列に格納されるので，一般にはその配列を以後の処理で使います．

例を示します．ファイルの中身が以下とします．

```
abc cba
efg h
```

以下を実行したとします．

```
char buf[128];
char *c;
c = fgets(buf,128,fp);
```

この結果，配列 buf に１行分の文字が入ります．具体的には，以下のようになります．

```
buf[0] = 'a', buf[1] = 'b', buf[2] = 'c', buf[3] = ' ',
buf[4] = 'c', buf[5] = 'b', buf[6] = 'a', buf[7] = '\n', buf[8] = '\0'
```

以下の 3 点がポイントです．

- buf[7] に改行の文字'\n' が入る．
- buf[8] に文字列の終りの記号'\0' が入る．
- buf[9] 〜 buf[127] に何が入っているかは不明．

fgets を実行すると，ファイル中の文字を示すポインタは，次の行の先頭（前の行の改行の次の文字）に移動します．

練習 13-3 練習 13-2 を fgets を使って書き直しなさい．

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    FILE *fp;
    char buf[128];        /* 1 行は 128 文字以下とする */

    if(argc != 2)  { printf("usage: prog filename\n"); exit(0); }
    fp = fopen(argv[1],"r");
    if (fp == NULL) { printf("cannot open file\n"); exit(0); }
    fgets(buf,128,fp);
    printf(buf);         /* 改行が必要ないことに注意 */
    fclose(fp);
}
```

13.4 ファイルからの１行の読み出し (fscanf)

1 行に 1 データの形で，データをテキストファイルに保管することはよく行われます．例えば，ファイル data.txt は以下のような中身だとします．

```
田中    18    170.2   65.5
佐藤    19    175.1   70.4
山田    18    164.0   55.7
```

１行は各人の名前，年齢，身長，体重を表しています．例えば，１行目は名前が「田中」で，年齢が 18 才，身長が 170.2 cm，体重が 65.5 kg を表します．そしてこのファイルは各行が以下のフォーマットに従っています．

```
名前<tab>年齢 (才)<tab>身長 (cm)<tab>体重 (kg)
```

この data.txt から平均年齢，平均身長，平均体重を求めるプログラムを書いてみます．ポイントはファイルから１行読んで，年齢，身長，体重を変数に代入することです．もちろん，先の fgets を用いてもできますが，ある種のデータの形式（後で説明します）では， fscanf という関数を使うのが簡単です．

fscanf は以下の形で使います．返る値は読み込んだ項目の数です． EOF を検出したときは EOF を返します．

```
int  r;
r = fscanf(ファイルポインタ, フォーマット, 変数 1, 変数 2, ..., 変数 n);
```

フォーマットの書き方，変数の書き方がポイントです.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    FILE *fp;
    char buf[128];        /* 1 行は 128 文字以下とする */
    char name[30];        /* 名前は全角で 15 文字以下 */
    int  person = 0;
    int  age, agesum = 0;
    double  h, hsum = 0.0;
    double  w, wsum = 0.0;

    if(argc != 2)  { printf("usage: prog filename\n"); exit(0); }
    fp = fopen(argv[1],"r");
    if (fp == NULL) { printf("cannot open file\n"); exit(0); }

    while(fscanf(fp,"%s\t%d\t%lf\t%lf",name,&age,&h,&w) != EOF) {
        person++;
        agesum += age;  hsum += h;  wsum += w;
    }
    fclose(fp);

    printf("平均年齢は %f\n",((double) agesum)/person);
    printf("平均身長は %f\n",hsum/person);
    printf("平均体重は %f\n",wsum/person);
}
```

[実行結果]

```
> prog data.txt
平均年齢は 18.333333
平均身長は 169.766667
平均体重は 63.866667
```

フォーマットの書き方は printf のファーマットと同じなので，すぐ分ると思います．\t はタブを表します．受ける方の変数は，アドレスを指定するのがポイントです．

fscanf は 1 行のデータの読み出しに利用できますが，本当の使い方はもう少し複雑です．まず，もとになったデータの data.txt のタブの部分が適当な数の半角の空白スペースになっていたとしたら，フォーマットの部分 "%s\t%d\t%lf\t%lf" は一致していないので，不適切に見えますが，上記のプログラムは正しく動きます．また上記のプログラムでフォーマットの部分を "%s %d %lf %lf" としても正しく動きます．更に， data.txt が以下の形のようにめちゃくちゃでも，正しく動きます．

```
田中
18   170.2   65.5
佐藤
19   175.1
70.4
山田                18
164.0
      55.7
```

理由は fscanf がファイルの 1 行ごとの処理をしていると考えるのが間違いなのです．空白，タブ，改行，これらは**ホワイト スペース文字**と呼ばれています． fscanf はホワイトスペース文字で区切られたデータを指定したフォーマットにしたがって，順に取り出しているのです．

fscanf が 1 行ごとの処理に使えるのは，データがホワイトスペース文字で区切られていることがポイントです．ホワイトスペース文字ではない文字で区切られていた場合，例えば，以下のように，カンマ"," で区切られていた場合はどうでしょう．

```
田中,18,170.2,65.5
佐藤,19,175.1,70.4
山田,18,164.0,55.7
```

実は少しテクニカルなことをすれば，このような場合も fscanf を利用できますが，いくつか注意しなければならない点が発生します．結局，バグが潜みやすいので，私は fscanf を勧めません．fgets でコツコツ書いていった方が確実です．

13.5 ファイルへの書き込み

ファイルへ書き込むときも，fopen でまずそのファイルを開く必要があります．また前回述べたように，ファイルを fopen で開くときにはモード（開く目的）を指定する必要があります．ファイルを読むときのモードは ”r” でしたが，書き込むときのモードは新規書き込み用の ”w” か，追加書き込み用の ”a” を使います．

例えば， fopen("prog.txt","w") で prog.txt を新規書き込みモードで開いた場合，まず prog.txt が既に存在していれば，それを消して，新しい空の prog.txt が作成され，それに書き込む形になります．つまり重要なファイルを消してしまわないように注意する必要があります．もし prog.txt が存在しなければ，新しくそのファイルをつくる形になります．

また fopen("prog.txt","a") で prog.txt を追加書き込みモードで開いた場合，まず prog.txt を開いて，そのファイルの最後の文字の後から追加して書き込む形になります．

ファイルに文字や文字列を書き込むには， printf のファイル出力版である fprintf を使います．以下の書式です．

```
fprintf(ファイルポインタ, ******);
```

***** の部分に書くのは， printf と同じ形式です．第 1 引数フのァイルポインタが指すファイルに ***** の部分に対応するデータを出力することになります．当然，第 1 引数のファイルポインタは "w" や "a" のモードで開かれたファイルに対するファイルポインタでなくてはいけません．

練習 13-4 適当な内容の書かれたファイル data.txt を作成する．このファイルを data2.txt へコピーするプログラムを作成しなさい．

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    FILE *fp,*fp2;
    char buf[128];

    fp = fopen("data.txt","r");
    fp2 = fopen("data2.txt","w");      /* 新規書き込みで開く */
    while(fgets(buf,128,fp) != NULL)  /* この条件式の形も大事 */
       fprintf(fp2,buf);               /* fp2 の指すファイルに buf 出力 */
    fclose(fp2);
    fclose(fp);
}
```

[実行結果]

```
> type data.txt
私は茨城大学の学生です

> prog

> type data2.txt
私は茨城大学の学生です
```

13.6 標準入力からの１行の読み込み

ファイルからの１行の読み込みには fgets という関数を使いました．標準入力からの１行の読み込みには fgets の姉妹版である gets という関数を使います[1]．

標準入力からの読み込みとして scanf という関数もあります．この関数は fscanf のファイルポインタが標準入力に限定されたものです． fscanf のところで注意した通り，使用はあまり勧めません．単純なプログラムなら gets だけで十分です．

練習 13-5 コンピュータがファイル名を聞く．利用者がファイル名を入力する．そしてそのファイルが開けるかどうかを調べて，その結果を表示するプログラムを作成しなさい．

```
#include <stdio.h>

int main()
{
    FILE *fp;
    char buf[128];

    printf("Please input file name\n");
    gets(buf);
    fp = fopen(buf,"r");
    if (fp == NULL) {
        printf("%s  can NOT be opened\n",buf);
    } else {
        printf("%s  can be opened\n",buf);
        fclose(fp);
    }
}
```

[実行結果]

```
> prog
Please input file name
data.txt
data.txt  can be opened
```

練習 13-6 利用者に数値を次々とキーボードから入れることを要求する．改行だけが入力されたら終り．そこまでに入力した数値の合計を表示するプログラムを作成せよ．

[1] UNIX 系のＯＳの１つの特徴は，デバイスもファイルとして扱えることです．そのため標準入出力もファイルの形で扱えます．つまり gets とは fgets の機能限定版と捉えることもできます．

実行イメージ

```
>prog
Input number
10
Input number
7
Input number
11
Input number

---
sum is 28

>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char buf[128];
    int sum = 0;

    printf("Input number\n");
    gets(buf);
    while(strcmp(buf,"") != 0) {
       sum += atoi(buf);
       printf("Input number\n");
       gets(buf);
    }
    printf("---\n");
    printf("sum is %d\n",sum);
}
```

13.7 リダイレクション

ファイルへの出力は DOS の**リダイレクション**の機能を利用しても可能です.

画面に出力させるプログラムがあり，その画面に出力させる出力をあるファイルに入れたいとします.

プログラム中の printf を fprintf などに変更して再コンパイルしてもできますが，リダイレクションの機能を使えばもっと簡単に実現できます．これは C 言語とは直接は関係ありません．OS の機能です.

```
> prog   （下のような結果を出力するプログラムとする）
1 + 2 + 3 = 6
1 + 3 + 2 = 6
2 + 1 + 3 = 6
3 + 1 + 2 = 6
2 + 3 + 1 = 6
3 + 2 + 1 = 6

> prog  > data.out     <== これがリダイレクション．画面への出力を
                           data.out へ出力させている．
>                      <== 何も結果は出力されない．

> type data.out        <== 結果は data.out に入っている．
1 + 2 + 3 = 6
1 + 3 + 2 = 6
2 + 1 + 3 = 6
3 + 1 + 2 = 6
2 + 3 + 1 = 6
3 + 2 + 1 = 6
```

13.8 パイプ

progA の出力を標準出力に出力し，その出力結果を progB で利用したい場合があります.

このときに**パイプ**という OS の機能が役に立ちます．基本的には以下の形で利用します.

```
> progA  |  progB    Enter
```

progA の出力は標準出力に出力されず，直接 progB に引き渡されます．progB の中では progA の出力は標準入力からの入力として扱われます.

例を示しましょう．練習 13-6 のプログラムを prac13-6.exe とします．次に以下のファイル data.txt を用意します.

```
> type data.txt
10
7
```

```
11
          <== 改行だけの行
[EOF]     <== ファイルの終わり，表示されない
```

次に以下のコマンドを実行してみて，何が起こるか確認しなさい．

```
> type data.txt | prac13-6.exe    Enter
```

演習問題１３

13-1 以下のように各行に整数だけを記したファイルがあるとします．このファイルを開いて，各行の数値の和を求めるプログラムを書きなさい．ファイル名はプログラムの引数にすること．

```
------ ファイル data.txt の中身 ------------
12
2
-4
6
321

---------------------------------
```

実行例

```
> prog data.txt
sum is 337
```

13-2 練習 13-4 を改良して，コピー元のファイル名とコピー先のファイル名をプログラムが尋ねるようにしなさい．

```
> prog
Please input FROM file
data.txt     <-- ここは利用者が入力
Please input TO file
data2.txt     <-- ここは利用者が入力

>
```

13-3 演習問題 13-1 のプログラムを，数値は標準入力から得るようにしなさい．そして，以下の命令で 演習問題 13-1 と同じ結果が出るようにしなさい．

実行例

```
> type data.txt | prog13-3
sum is 337
```

第14章　構造体

14.1　構造体の定義

同じデータ型のデータを複数保存するために配列というものがありました．**構造体**とは異なるデータ型のデータを複数保存するためのものです．

配列では含まれる複数のデータにインデックス（配列の要素番号）をつけ，そのインデックスから各データにアクセスできました．構造体の場合は，それぞれのデータに**メンバ名**（**フィールド名**とも呼ばれる）をつけて，そのメンバ名を介してアクセスできます．

構造体は「異なるデータ型のデータを複数保存するためのもの」と述べましたが，別に異なるデータ型でなくてもかまいません．同じデータ型だと本質的に配列と同じですが，各データにインデックスではなく，メンバ名をつけられるので，分かりやすいプログラムになることもあります．

構造体の定義の一般形式は以下のとおりです[1]．

```
struct  構造体名  {
    データ型    メンバ名;
    データ型    メンバ名;
        ....
    データ型    メンバ名;
};
```

最後の ";" は要らないようにも見えますが必要です．注意してください．

例を示します．ある学生のデータとして，その学生の名前，学年，取得単位の３つがあるとします．この学生のデータは構造体を利用して以下のように表せます．

```
struct  student  {
    char        name[30];
    int         grade;
    int         credits;
};
```

ここで student は構造体の名前です．struct student の２単語で int や char と同じデータ型となります．

[1] この定義は**無名構造体**と呼ばれます．もっと便利な形もありますが，本書ではこの形だけで通します．

つまり struct student { 〜 } というのは，struct student という型を定義したものと見なすことができます．型を定義した後は，この型を持つ変数を定義することができます．以下では，struct student 型の変数として abc を定義しています．

```
struct student abc;
```

14.2 メンバへの参照と代入

先の例では変数 abc の型は構造体 student でした．なので，name，grade，credits というデータを保持できます，あるいは保持しています．それらのデータへのアクセスは，以下の形で実現できます．

変数名 . メンバ名

例えば，変数 abc の name は，abc.name でアクセスできます．abc.name は一つの変数名であり，char 型の要素数 30 の配列となっています．ですので，例えば，変数 abc の name の第 1 文字目は abc.name[0] で参照できます．

代入は通常の代入と同じです．

練習 14-1 構造体 student を定義し，abc を構造体 student 型の変数とする．abc の name に "suzuki"，grade に 2，credits に 128 を代入し，それらを表示するプログラムを書きなさい．

```
#include <stdio.h>
#include <string.h>

struct  student  {              /*-------------------------- */
    char        name[30];       /*  ここ大事，このように     */
    int         grade;          /*  型は main の前で宣言する */
    int         credits;        /*                            */
};                              /*--------------------------*/

int main()
{
    struct student abc;

    strcpy(abc.name,"suzuki");
    abc.grade = 2;
    abc.credits = 128;

    printf("名前は %s \n",abc.name);
    printf("学年は %d \n",abc.grade);
    printf("単位は %d \n",abc.credits);
}
```

[実行結果]

```
名前は suzuki
学年は 2
単位は 128
```

宣言時に初期化する場合には，もっと簡単に以下のように書けます.

```
#include <stdio.h>
#include <string.h>

struct  student  {
    char       name[30];
    int        grade;
    int        credits;
};

int main()
{
    struct student abc = {"suzuki", 2, 128};  /* ここ注意 */

    printf("名前は %s \n",abc.name);
    printf("学年は %d \n",abc.grade);
    printf("単位は %d \n",abc.credits);
}
```

14.3 構造体の配列

先ほど定義した構造体 student が一つのデータ型であることをふまえれば，構造体の配列は容易に実現できます．例えば，構造体 student のデータを 40 個分用意した変数 acls は以下のように定義できます．40 名分のデータが取り扱えます．

```
struct student acls[40];
```

練習 14-2 構造体 dpos は 2 次元格子平面上の点 (x,y) を表し，以下の形で定義される．

```
struct  dpos  {
    int    x;
    int    y;
};
```

構造体 dpos 100 個からなる配列 hdata を定義し，各要素の構造体 dpos のメンバ x と y に 100 以下のランダムな値を代入しなさい．その後で，$x + y$ の値が最も大きい要素を表示しなさい．

```
#include <stdio.h>
#include <stdlib.h>

struct  dpos {
   int x;
   int y;
};

int main()
{
    struct dpos hdata[100];
    int i,sum,max,maxpos;

    for(i = 0;i <100;i++) {
       hdata[i].x = rand()%100;
       hdata[i].y = rand()%100;
    }

    maxpos = 0; max = 0;
    for(i = 0;i <100;i++) {
       sum = hdata[i].x + hdata[i].y;
       if(max < sum) {
         max = sum; maxpos = i;
       }
    }

    printf("Max is %d ( = %d + %d )\n",
           max,hdata[maxpos].x,hdata[maxpos].y);
}
```

[実行結果]

```
Max is 178 ( = 86 + 92 )
```

14.4 構造体を関数へ渡す

先ほど定義した構造体 student が一つのデータ型であることをふまえれば，構造体へのポインタは容易に実現できます．例えば，構造体 student の変数 abc へのポインタは，通常の変数のポインタと同様，以下の形で実現できます．

```
&abc
```

変数 a が構造体 student へのポインタである場合，構造体のメンバにアクセスするには，*a が構造体の変数となることを考えれば，(*a).name などとすれば良いことは明らかです．

関数に構造体を渡す場合，配列と同様，そのポインタを渡せばよいです．具体的には構造体の変数のポインタを使えばできます．

例） `funct(&abc)`

関数の定義は以下のように行います．この例では返る値の型は int としています．

例） `int funct(struct student *a)`

これは渡される変数が関数内では変数 a でアクセスでき，しかも変数 a は構造体 student へのポインタであることが宣言されています．

関数のプロトタイプ宣言の部分は，以下の形となります．

例） `int funct(struct student *)`

練習 14-3 構造体 student のデータを表示する関数 print_student を作成しなさい．

```
#include <stdio.h>
#include <string.h>

struct  student  {
    char       name[30];
    int        grade;
    int        credits;
};

void print_student(struct student *);
          /* 注意, 構造体 student の定義の後に宣言しなければならない */

int main()
{
    struct student abc;

    strcpy(abc.name,"suzuki");
    abc.grade = 2;
    abc.credits = 128;

    print_student(&abc);
}

void print_student(struct student *a) {
    printf("名前は %s \n",(*a).name);
    printf("学年は %d \n",(*a).grade);
    printf("単位は %d \n",(*a).credits);
}
```

[実行結果]

```
名前は suzuki
学年は 2
単位は 128
```

上のプログラムでは (*a).grade のような書き方がされています. これを*a.grade とは書けません. 意味が異なります. *a.grade は a.grade という変数がポインタであり, そのポインタの指すデータを表します.

(*a).grade のような書き方は少し煩雑なので, 同じ意味で a->grade という書き方があります. こちらの方が一般に使われています.

```
void print_student(struct student *a) {
    printf("名前は %s \n",a->name);
    printf("学年は %d \n",a->grade);
    printf("単位は %d \n",a->credits);
}
```

14.5 システムが提供している構造体

システムで用意されている少し高級な関数を使おうとすると，多くの場合，その引数や返り値のデータがその関数用に作られた構造体になっていますシステムで用意されている関数を使いこなすには，そういった構造体を調べる必要があります.

システムが定義した構造体は，あるヘッダーファイルに記載されています．その構造体を利用する場合は，その構造体が定義されているヘッダーファイルをインクルードする必要があります.

例えば，stdlib.h では div という関数が定義されています．この関数は 2 つの整数を引数にとり，第 1 引数を第 2 引数で割ったときの商と余りを返す関数です．商と余りを同時返すために，その 2 つのデータを 1 つにまとめた div_t という構造体を新たに定義しています．div_t は以下のメンバーを持つ構造体です.

```
struct div_t {
   int quot;
   int rem;
};
```

結局，関数 div の返り値の商の部分を quot に，余りを rem に入れて，div_t 型のデータを返します．以下が利用例ですが，使い方は少し難しいです．div_t の変数を宣言する部分で使います.

```
#include <stdio.h>
#include <stdlib.h>  /* div_t と div の定義で必要 */

int main()
{
    div_t dt = div(3277,121);
    printf("3277 / 121 = %d ... %d\n", dt.quot, dt.rem);
}
```

[実行結果]

```
3277 / 121 = 27 ... 10
```

演習問題１４

14-1 構造体 student のデータを二つ用意し，それらをコピーする関数 copy_student を作りなさい．

14-2 ファイル data.txt には１行毎に名前と年収が記載されている．名前と年収の間には適当に空白文字が入っている．20 名分入っているとする．例えば以下のような感じです．

```
> type data.txt
suzuki      212
yamada      110
mori       1200
tanaka       50
...
matumoto    331
```

ファイル data.txt を引数にして，年収順に表示するプログラムを書きなさい．実行例は以下のような感じです．

```
> prog data.txt

mori       1200
kiyohara    532
matumoto    331
...
tanaka       50
```

プログラムの中では１人に対して，以下の構造体 person のデータを割り当てること．20 名分のデータはこの構造体の配列を利用せよ．ソートのアルゴリズムは何でも良い．

```
struct  person  {
    char        name[30];
    int         salary;
};
```

14-3 time.h で定義されている time_t 及び tm という構造体を調べなさい．またそれらを使った関数 time や localtime を用いて，今日が何曜日かを表示するプログラムを作成しなさい．

第15章　動的メモリの確保

15.1　動的メモリとは

プログラムを作っているとデータの数が予め決められないような問題が多々あります．例えば，ファイルから複数の整数を読み取り，それらをソートして出力することを考えます．今まで学んだ方法では，データの整数値をすべて配列に入れておく必要がありました．しかし問題のファイルには幾つデータがあるかは，現実には，ファイルを開いてみないとわかりません．ですので，非常に大きな配列を用意しなければなりません．

```
int data[1000];
```

しかし，実際のデータ数は 20 や 30 の場合もあり，その場合，大きな配列をとったのは無駄でした．また逆に非常に大きな配列を用意したが，それには収まらないようなデータが入力されることもありえます．

こういった場合に，**動的メモリ**の確保が使われます．

動的メモリの確保とは，プログラムの実行中に主記憶上に新たに作った変数のメモリを確保する方法です．int 型のような単純なものはほとんど使うことはありませんが，構造体では比較的よく使います．

プログラムの中ではまずその構造体へのポインタだけを定義しておきます．ポインタだけなので，実体はメモリ上には存在しません．必要に応じて，メモリを確保して，その確保したメモリのアドレスを先のポインタの値にします．

大事な点は以下の二つです．

- どの程度のメモリを取ればよいのか？
- どうやってメモリを確保すれば良いのか？

15.2　データ型のサイズ

どの程度のメモリを取れば良いかは簡単です．そのデータの型が要求するサイズを取れば良いです．

データ型がどのくらいのサイズのメモリを要求するかは sizeof という関数で取得できます．結果の数値の単位はバイトです．

以下の例を実行すれば，double 型のサイズ（バイト数）がわかります．私の環境では 8 バイトになりました．

```
int main() {
   printf("%d\n",sizeof(double));
}
```

sizeof は構造体のサイズを知るのにも使われます．以下の例では，私の環境では 40 バイトになりました．

```
struct  student  {
    char       name[30];
    int        grade;
    int        credits;
};

int main() {
   printf("%d\n",sizeof(struct student));
}
```

15.3 メモリの確保

メモリの確保には malloc 関数を使います．これは引数で指定されたバイト数だけ主記憶上にそのメモリを確保します．そしてその確保したメモリの先頭番地を返します．返す値はポインタの値となりますが，どの型のポインタかを指定する必要があります．これにはキャストを用います．

例えば，double 型のデータを 1 個確保する場合は，以下のようになります．

```
double *pt;
pt = (duble *)malloc(sizeof(double))
```

いきなり 10 個確保しても構いません．

```
double *pt;
pt = (double *)malloc(sizeof(double)*10)
```

この場合，2 番目のデータには *(pt + 1) でアクセスできます．

練習 15-1 構造体 student へのポインタとなる変数 abc を定義し，プログラムの中でその実体のメモリを確保して，name に "suzuki"，grade に 2，credits に 128 を代入し，それらを表示するプログラムを書きなさい.

```
#include <stdio.h>
#include <stdlib.h>  /* malloc も stlib.h */
#include <string.h>

struct  student  {
    char       name[30];
    int        grade;
    int        credits;
};

int main()
{
    struct student *abc;

    abc = (struct student *)malloc(sizeof(struct student));

    strcpy(abc->name,"suzuki");
    abc->grade = 2;
    abc->credits = 128;

    printf("名前は %s \n",abc->name);
    printf("学年は %d \n",abc->grade);
    printf("単位は %d \n",abc->credits);
}
```

[実行結果]

```
名前は suzuki
学年は 2
単位は 128
```

15.4 確保したメモリの解放

malloc で確保したメモリは，プログラムが動いている間，主記憶上に居座ります. 次々に malloc でメモリを確保すると，そのうちメモリを使いきり，プログラムは異常終了します.

確保したメモリが不要になったら，その段階でそのメモリを解放する必要があります．メモリを解放するには free という関数を使います．使い方は以下の通りです．

```
double *pt;
pt = (double *)malloc(sizeof(double)*10)

free(pt);  /* 先に確保した 10 個分を解放する */
```

練習 15-2 自分のシステムで malloc をつかって，double 型のデータを幾つまで確保できるかを調べるプログラムを書きなさい．また free を使った場合はその制限を越えられることを確認しなさい．

●幾つまで確保できるかを調べるプログラム

```
int main()
{

   int i = 0;
   double *pt;

   pt = (double *)malloc(10000*sizeof(double));
   while(pt != NULL) {
      i++;
      printf("%d 万個確保\n",i);
      pt = (double *)malloc(10000*sizeof(double));
   }
}
```

[実行結果]

```
1 万個確保
2 万個確保
3 万個確保
...
```

● free を使ったプログラム（適当に強制終了させること）

```
int main()
{
   int i = 0;
   double *pt;

   pt = (double *)malloc(10000*sizeof(double));
   while(pt != NULL) {
      i++;
      printf("%d 万個確保\n",i);
      free(pt);
      pt = (double *)malloc(10000*sizeof(double));
   }
}
```

[実行結果]

```
1 万個確保
2 万個確保
3 万個確保
...
```

「確保したメモリはそのプログラムが終了しても主記憶上に残るので，確保したメモリは必ず free で解放する必要がある」というのは，ほとんどの場合当てはまらりません．利用している OS に依存します．また現実的に free を使う場面はほとんどないはずです．

15.5 複雑なデータ構造の実現

プログラミングで重要なデータ表現形式として，**ツリー**や**リスト**があります．

ここでは詳しく述べませんが，そこでは構造体のメンバにその構造体へのポインタが使われます．例えば，以下はリストのノードを表す構造体です．

```
struct  node  {
    int             value;
    struct  node    *next;  /* 再帰的に node の定義を使っている */
};
```

上記の struct node のデータ型の変数を宣言した場合，メンバ value に対して sizeof(int)

のメモリ，メンバ next に対して sizeof(struct node *) のメモリが確保されるのであって，決してメンバ next に対して sizeof(struct node) のメモリが確保されるわけではありません．

そのため実際にメンバ next に新たなノードをつなげたい場合には，malloc で sizeof(struct node) のメモリを確保する必要があります．

演習問題１５

15-1 上記の構造体 node を利用して，ランダムに生成した 100 個の 1000 以下の整数値を，リスト構造で実現しなさい．

15-2 メンバ名 next の接続先を変更することで，演習問題 15-1 のリストを数値の大きい順にソートするプログラムを書きなさい．

第16章　関数を引数とする関数

C 言語では関数をポインタとしてして関数に渡すことができます．このテクニックを使うとより汎用的な関数を作ることができます．例えば，構造体のデータをソートすることを考えてみます．構造体のどのメンバーに注目して，どのようなオーダー（大きい順，小さい順，辞書順など）によってソートするかでソートのプログラムは変化します．ところが，構造体どうしの比較を一つの関数として定義し，ソートの関数にはその比較の関数を渡すようにすれば，ソートアルゴリズムの核になる部分のプログラムは変化なしにそのまま使えます．

C 言語の標準ライブラリには `qsort` というクイックソートを用いたソートのプログラムがありますが，そこでは比較の関数を引数として与えています．

16.1　記述例

問題は記述方法です．まず簡単な例を示します．

練習 16-1　２つの整数の足し算の関数 tasu，引き算の関数 hiku，掛け算の関数 kakeru，割り算の関数 waru を引数にできる関数 funct を作りなさい．

```
#include <stdio.h>

int funct(int (*)(),int,int);  /* ここがポイント */

int tasu(int,int);
int hiku(int,int);
int kakeru(int,int);
int waru(int,int);

int main()
{
    printf("足し算 %d\n", funct(tasu,7,3));
    printf("引き算 %d\n", funct(hiku,7,3));
    printf("掛け算 %d\n", funct(kakeru,7,3));
    printf("割り算 %d\n", funct(waru,7,3));
}

int funct(int (*p)(), int x, int y)   /* ここがポイント */
{
    return (*p)(x,y);                 /* ここがポイント */
}

int tasu(int x,int y)
{
   return x + y;
}

int hiku(int x,int y)
{
   return x - y;
}

int kakeru(int x,int y)
{
   return x*y;
}

int waru(int x,int y)
{
   return x/y;
}
```

[実行結果]

```
足し算 10
引き算 4
掛け算 21
割り算 2
```

16.2 汎用のポインタ

前述した例は単純です．なぜなら引数とされる関数 tasu ，hiku，kakeru，waru の引数の数，引数の型，返り値の型が一致しているからです．これらを可変にするにはいろいろテクニックが必要です．ここでは引数の型が違う場合の扱いだけを述べます．引数の数と返り値は同じとします．結論を言えば以下の型を利用します．

```
const void *
```

これは**汎用のポインタ**と呼ばれているものです．

練習 16-2 ２つの整数の足し算の関数 tasu，引き算の関数 hiku，及び２つの実数の掛け算の関数 kakeru，割り算の関数 waru を引数にできる関数 funct を作りなさい．ただしそれらの関数の返り値の型はすべて double 型とする．

まず前半部は以下のようになります．

```
#include <stdio.h>

double funct(double (*)(),const void *,const void *);

double tasu(const void *,const void *);    /*---------------------*/
double hiku(const void *,const void *);    /*  引数の型は全部     */
double kakeru(const void *,const void *);  /*  汎用のポインタ     */
double waru(const void *,const void *);    /*---------------------*/

int main()
{
    int a = 7, b = 3;
    double c = 7.0, d = 3.0;

    printf("足し算 %f\n",funct(tasu,&a,&b));
    printf("引き算 %f\n",funct(hiku,&a,&b));
    printf("掛け算 %f\n",funct(kakeru,&c,&d));
    printf("割り算 %f\n",funct(waru,&c,&d));
}

double funct(double (*p)(),const void *x,const void *y)
{
    return (*p)(x,y);
}
```

引数になる関数の定義は以下のようになります.

```
double tasu(const void *x,const void *y)
{
   const int *a = x;   /* このように取り出せば良い */
   const int *b = y;
   return (double)(*a + *b);
}

double hiku(const void *x,const void *y)
{
   const int *a = x;
   const int *b = y;
   return (double)(*a - *b);
}

double kakeru(const void *x,const void *y)
{
   const double *a = x;  /* このように取り出せば良い */
   const double *b = y;
   return (*a)*(*b);
}

double  waru(const void *x,const void *y)
{
   const double *a = x;
   const double *b = y;
   return (*a)/(*b);
}
```

[実行結果]

```
足し算 10.000000
引き算 4.000000
掛け算 21.000000
割り算 2.333333
```

16.3 qsort の利用

qsort は C 言語の標準ライブラリとして提供されているソートのプログラムです．ソートアルゴリズムとして**クイックソート**を用いており高速です．

qsort は４つの引数を持ちます．返り値は void です．使用するときは， stdlib.h をインクルードするので，プロトタイプの宣言は必要ありません．呼び出し方が問題です．以下のように呼び出します．

```
qsort(配列名，配列の要素数，配列の１つのサイズ，比較関数名);
```

qsort は第 1 引数で示された配列の要素を，第 4 引数で示された比較関数で比較して，小さい順にソートします．第 1 引数の配列は何の配列でも構いません．整数の配列であれ，構造体の配列であれ，文字列の配列であれ，何でもＯＫです．

比較関数を今 cmpfn としておきます．cmpfn は以下の形式の関数です．

```
int cmpfn(汎用のポインタ *e1，汎用のポインタ *e2)
{
   ...
}
```

e1 と e2 に比較すべき値へのポインタが汎用のポインタとして渡されます．この場合，具体的には配列の要素へのポインタとなっています．結局 qsort の場合の比較関数の汎用のポインタというのは，配列の要素へのポインタとなります．そして，e1 の実体が e2 の実体よりも小さい場合には負の値を返し，e1 の実体が e2 の実体よりも大きい場合には正の値を返し，等しい場合は 0 を返すように作ります．

練習 16-3 ランダムに生成した 1000 以下の 100 個の整数を標準ライブラリの qsort を用いて，小さい順に並べるプログラムをかきなさい．

```
#include <stdio.h>
#include <stdlib.h>

int cmpfn(const void *,const void *);

int main()
{
   int i,ary[100];

   for(i = 0;i < 100;i++) ary[i] = rand()%1000;

   qsort(ary,100,sizeof(int),cmpfn);  /* ここの書き方がポイント */

   for(i = 0;i < 100;i++)
     printf("%i 番目に大きい数は %d です\n",i,ary[i]);
}

int cmpfn(const void *e1,const void *e2)
{
   const int *a = e1;
   const int *b = e2;

   if (*a < *b) return -1;
   if (*a > *b) return 1;
   return 0;
}
```

[実行結果]

```
0 番目に大きい数は 11 です
1 番目に大きい数は 12 です
2 番目に大きい数は 22 です
...
97 番目に大きい数は 956 です
98 番目に大きい数は 980 です
99 番目に大きい数は 996 です
```

qsort の第３引数は，配列の要素のデータ型のサイズ数なので，sizeof 関数を使います．また上記の cmpfn はもっと単純に以下のようにも書けます．

```
int cmpfn(const void *e1,const void *e2)
{
   const int *a = e1;
   const int *b = e2;

   return (*a)-(*b);
}
```

16.4 ライブラリの利用と作成

プログラムの生産性をあげる有効な方法は，既に作らているプログラムを利用することです．大規模なプログラムは 1 人で作れるものではありません．多くの人がいろいろな部分のプログラムを作り，それをまとめて大規模なプログラムができ上がります．まとめる人から見れば，多くの人が作った個々のプログラムを利用して（大規模な）プログラムを作ったことになります．

再利用できそうなプログラムは，多くの場合，ライブラリという形で提供されています．自分が作りたいプログラムがあれば，まず利用できるライブラリがないかどうかを調べるのも一方法です．既に説明しましたが，C コンパイラには標準ライブラリというものがついています．それらのライブラリを自由に使えるようになるだけでも，相当，プログラミングの効率は上がります．

また自分で作る関数も再利用を考えるならば，できるだけ汎用的な形にすべきです．そしてライブラリにして他の人に使ってもらえるようになれば最高です．そのためには関数を引数とする関数を作る必要もあるでしょう．そのために，ここで章をとってこの手法を説明しました．

整数の入った配列の要素をクイックソートでソートするプログラムは単純です．しかし単純に作成されたプログラムは,「配列に入った整数値」のソートしか行えません．もしもファイルに書かれた名前のリストを辞書順にクイックソートするなどといった問題を考えた場合，再び，プログラムしなくてはなりません．クイックソートのアルゴリズムは「配列に入った整数値」だけにしか使えないわけではありません．あるオーダーをもとにデータを並べるという処理一般に使えるはずです．その点で前章で使ったクイックソートのライブラリ qsort には学ぶべき点が多くあります．再利用できるプログラムを作ることは大事です．

演習問題１６

16-1 演習問題 14-2 のソートの部分を標準ライブラリ qsort を用いて書き換えなさい．

第17章　その他知っておいた方がいいこと

本書はとりあえずプログラムできることを目指しました．説明していない重要な点も多く残っています．説明を落した部分について簡単に触れておきます．

17.1　大域変数

大域変数とはプログラム中のどの関数の中からでも参照できる変数です．一般には大域変数を使うことは推奨されていません．ただ使った方が良い場合もあることはたしかです．

大域変数の使い方は，プログラムを複数のファイルに分割しているか，それとも自分が定義したすべての関数を１つのファイルに書いているかによって多少異なります．ここでは後者の場合を説明します．

関数のプロトタイプの後に，以下のように宣言しておけば大域変数が使えます．

```
int glvar = 105;  /* 例として変数名は glvar とします */
```

以下に例を示します．

```
#include <stdio.h>

void f1(void);
void f2(void);

int glvar = 10;

int main()
{
    printf("glvar = %d\n",glvar);
    f1();
    printf("glvar = %d\n",glvar);
    f2();
    printf("glvar = %d\n",glvar);
}

void f1(void)
{
   glvar++;
}

void f2(void)
{
   glvar = 1;
   f1();
}
```

[実行結果]

```
glvar = 10
glvar = 11
glvar = 2
```

大域変数はどの関数からでも変更が可能です．複数人でプログラムを開発していると，自分が作っている関数では，ある値だと思ってその大域変数を使っても，別の人が作っている関数で勝手にムチャクチャな値を入れられることもあります．大域変数を使うときは注意が必要です．

17.2 スタティック変数

ある関数 funct がプログラム中で何回呼ばれたかを調べたいとします．基本的には大域変数 count を用意し，プログラムの中の funct が呼ばれるすべての個所で， `count++;` を挿入すれば実現で

きます．ただ前述した通り，あまり大域変数を使うことは推奨されていません．このような場合には**スタティック変数**（static 変数）を使います．関数内の変数は一般には呼び出されるごとに初期化され，その関数を抜けると，それら変数の値は利用できません．static 変数はその関数にとどまる変数です．

```
#include <stdio.h>

void funct(void);

int main()
{
    funct(); funct(); funct();
}

void funct(void)
{
   static int i = 1;

   printf("%d 回呼ばれました\n",i);
   i++;
}
```

この結果は以下のようになります．

[実行結果]

```
1 回呼ばれました
2 回呼ばれました
3 回呼ばれました
```

大域変数に static が付くと，それは static 変数とは全く別の意味となることに注意してください．大域変数に static が付くと，その大域変数はそのファイル以外からは見ることができないことを意味します．なので，ファイルを分割してプログラムしなければ関係ないことです．

また関数にも static をつけることができます．以下のような感じです．

```
static void funct(void) {

}
```

こうすると，この関数 funct はそのファイル以外からは見えなくなります．これも，ファイルを分割してプログラムしなければ関係ないことです．

static を付けて，わざわざ外から見えなくすることにどんな意味があるのか？これはプログラムの情報隠蔽と呼ばれるテクニックです．詳しくは述べません．大規模なプログラムを作る際に必要となります．

17.3 符合付きの型

厳密には整数型には以下の４つの型があります[1]．

```
signed char   ==>  char    と略される
short int     ==>  short   と略される
int
long  int     ==>  long    と略される
```

そしてこの４つの型について，それぞれ**符合なしの型**があります．

```
unsigned char
unsigned short int    ==>  unsigned short    と略される
unsigned int
unsigned long  int    ==>  unsigned long     と略される
```

違いは表せる数値の範囲です．

実数型は以下の３つの型があります．

```
float
double
long double
```

17.4 switch 文，do 〜 while 文，goto 文，?:文

switch 文という条件分岐の文もあります．便利ですが間違いやすいです．if 文を使えば十分です．

do 〜 while 文も while 文や for 文を使えば良いです．

goto 文がどうしても必要になることはめったにありません．初心者のうちは使わない方が良いでしょう．

クエスチョン（?）とコロン（:）を使って， `if-then-else` の文が書けます．簡単にかけるので多用する人がいますが，きちんと if 文を使う方が良いです．

[1] 本書では char は文字型として説明したが，整数としても扱えるために，整数型ととして捉える人もいる．

17.5 ビット演算子

例えば整数型の変数はコンピュータ内では 32 ビットで表されています．その 32 個の各々のビットを操作する演算子がビット演算子です．以下のようなものがあります．

&	ビットごとの and
|	ビットごとの or
^	ビットごとの xor
~	１の補数（各ビットの反転）
<<	左シフト
>>	右シフト

特にこれらを使わなければならないという場面はないと思います．逆に

```
if ((a != 1) && (b == 2)) { ... }
```

を以下のように書き誤ったりしてもコンパイルは通るので，バグ探しが面倒になります．注意のために，こんな演算子もあることを示しました．

```
if ((a != 1) & (b == 2)) { ... }
```

17.6 マクロ

例えば配列 ary の大きさを 100 と固定したプログラムを作った後に，しばらくして，ary の大きさを 200 に変更したくなったとします．このとき，単純に ary の宣言部分だけを書き直せばよいというわけでありません．例えば，以下のような文がどこかにあるかもしれません．

```
for(i = 0;i < 100;i++) ary[i] = 1;
```

このような事態を避けるために，**マクロ**というものが用意されています．プログラムの最初の方に以下のようにして，配列の大きさを ARY_MAX という名前をつけておけば，後で変更が容易です．

```
#define ARY_MAX  100
```

プログラム全体は以下のような感じになります．プログラムの中で数値が直接見えるのは許せないという人たちもいます．マクロを使って，できるだけ数値は変数化した方が保守性が高いです．

```
#include <stdio.h>

#define ARY_MAX  100

 ....

int main() {

   int  ary[ARY_MAX];

   for(i = 0;i < ARY_MAX;i++) ary[i] = 1;

   ....

}
```

また定数として予めきまっているものはマクロとして既に与えられています. 例えば, M_PI は円周率であり, M_E は自然対数の底です.

```
#include <stdio.h>
#include <math.h>  /*  M_PI や M_E が #define されている */

int main()
{
   printf("円周率は %lf\n",M_PI);
   printf("自然対数の底は %lf\n",M_E);
}
```

[実行結果]

```
円周率は 3.141593
自然対数の底は 2.718282
```

マクロには上記のような単純なもの以外に, **関数形式マクロ**と呼ばれるマクロがあります. 例えば以下のようなものです.

```
#define nijou(a) ((a)*(a))
```

これを以下のように使ったとします.

```
x = nijou(5);
```

これはコンパイル前に以下のように置換されます.

```
x = ((5)*(5));
```

これによって所望の計算を行おうとするものです.

ただし，関数形式マクロを使う必要は全くありません．関数を書けば済むことです．関数形式マクロはいくつかワナがあって，間違いやすいので，使わない方が良いでしょう.

17.7 共用体と列挙型

どちらもキーワードとして知っているだけでよいと思います．正直，私は共用体や列挙型を使った（練習問題以外の）プログラムを書いたことがありません.

17.8 分割コンパイルと make

ここまでのプログラミングは，おそらく，1つのファイルに定義する関数を全部書いて，そのファイルをコンパイルして実行ファイルを作るという形でやってきたと思います.

ただこのような開発が可能なのは，プログラムの規模が小さい場合に限られます．大規模なプログラムを作成する場合には，プログラムを幾つかのファイルに分割し，それぞれのファイルをコンパイルして，最終的にはそれらのコンパイルされたファイルを結合して，実行ファイルを作ります.

大規模なプログラムを作るときは，プログラムの書かれたファイルを分割すべきです．もしも1つのファイルに書いていたら，たった1個所の修正を行う度に，プログラム全体をコンパイルしなおさなくてはなりません．分割しておけば，修正個所の影響があるファイルだけを再コンパイルするだけですみます.

ここで問題があります．ある修正を行ったときに，どのファイルを再コンパイルしなくてはならないかは，よく調べてみないと分りません．このような背景から，作られたツールが make です[2]. まず Makefile というファイルにファイル間の依存関係を定義しておきます．make はそのファイルをもとに，変更のあったファイル及びそれに伴って再コンパイルが必要なファイルだけを再コンパイルします．開発ツールとして大事です.

17.9 デバッガ

プログラムを書いて，コンパイルして，実行ファイルが作成できれば，そのプログラムはひとまずは文法的なエラーはなかったことになります．しかしそれだけでプログラムが正しく動くとは限

[2]Borland C++ Compiler 5.5 にも make.exe が付いています.

りません．動かしてみると全く動かなかったり，途中で止まったり，とんでもない結果が表示されたりと，予期せぬ事態がおこる場合がほとんどです．ここからデバッグという作業が始まります．

デバッグを行う最も単純な方法は，プログラムの中に適当に printf 文を入れて，その時点での変数の値などを表示させることです．このような printf 文は**デバッグライト**と呼ばれています．

しかし世の中には**デバッカ**というすばらしいツールが存在します．デバッカがあれば，**ブレークポイント**を設けて，その地点で実行を中断させたり，１行づつ**ステップ実行**したりできます．そのときの変数なども参照できます．そのためにどこに間違いがあるのかを容易に発見することができます．Borland C++ Compiler 5.5 に対しては Turbo Debugger 5.5 というデバッカが存在します．これも Borland のホームページから無料で入手可能なので，入手を勧めます．

Turbo Debugger 5.5 の使い方はまずプログラムをコンパイルする際に，`-v` というオプションをつけます．

```
> bcc32 -v myprog.c    Enter
```

そして出来た実行ファイルを対象にデバッグを行います．

```
> td32 myprog    Enter
```

ここからの詳しい使い方はここでは述べられませんが，HELP が付いているので，少し動かしていれば，自分にとって役立つ機能は自然に分ってきます．

おわりに

こんなテキストに「おわりに」など書くこともありませんが，今後のみなんさんのプログラミングの学習について，私が大事だと思っていることを２点述べて終わりにしたいと思います．

- 一度大きなプログラムを作ってみる．

私は大学生の時，授業で出された課題を解くのに，はじめて 1000 行を超えるプログラムを書きました．それは私にとって非常に有益な体験でした．以後，プログラムはまったく苦にならなくなりました．一度大きなプログラムを自力で作ってみると，そのプログラム言語のことは急激にわかるようになります．自分のパターンのようなものも出来上がります．

- ライブラリを使う技術を身につける．

どのようなプログラム言語であれ，その全体を理解するのは適当な時間さえかければ可能です．C 言語の文法などたいしたことはありません．しかし大規模なプログラムを作るには，それだけでは不十分です．他人が作ったプログラムを利用する技術が必要です．具体的には，本書でも繰り返し述べましたが，ライブラリを使う技術です．

Visual C++ という言語があります．多くの人に使ってもらうソフトを開発するにはこの言語をマスターする必要があるかもしれません．C++ 自体を理解するのは何の問題もないでしょう．しかしそれだけでは，この言語は使いこなせません．MFC ライブラリや API 関数の理解が必要となるからです．それだけでも相当な数ですが，世の中には独自のライブラリやツールなども存在します．それらを全部使いこなすのは不可能です．

ライブラリを使う技術を身につけるとは，コツコツと情報収集を行い，使えるライブラリを増やすことです．結局そうした作業を行えた人が,「プログラムのできる人」ということになると思います．

索 引