プログラミング演習 A 教材 (#1)

■ 概要説明

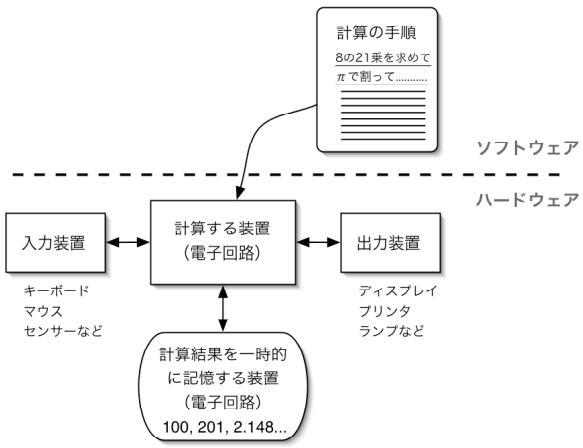
この科目では C 言語によるプログラミングについて学びます。一通りの C 言語についての理解と、操作方法を習得することが目標です。まずこの学期で、簡単なプログラムが組めるようになることが重要です。
実習は UNIX OS 上で C 言語を用いて行います。

この単元ではまず実際にプログラミングに至るまでに必要となる知識について説明します。多くの点については他のクラスでより詳しい説明がありますので、ここでは簡単にまとめています。

■ 予備知識

□ コンピュータについて

コンピュータとはある種の自動計算機械です。その内部は単純な計算を行う電子回路からできています。いわゆる電卓も計算を行う電子回路からできている製品ですが、電卓とコンピュータの本質的な違いは、自動処理についてだと考えて良いでしょう。



コンピュータには一連の計算を手順に沿って順序よく実行し、結果を残す機能があります。残された途中結果や、センサーからの情報によって幾つかある計算処理のいずれかを選んだり、たくさんのデータ（計算対象となる数値の列）に対して順繰りに同じ計算処理を行うようなことができます。
典型的なコンピュータ(*1)を機械として眺めてみると、図 1. のようになります。

それぞれの装置がどのような役割を果たすのかはこのクラスでは説明しません(*2)が、ともかくコンピュータの中身（図の下半分）が電子回路または機械装置で実現されていることに注目してください。これをハードウェアと呼んでおり、コンピュータの「計算する」という機能を実現している部分にあたります。
コンピュータの特徴は自動処理にある、と言いましたが、この自動化を担うのがソフトウェアです。ソフトウェアの実体は計算手順の列です。これをプログラムと呼んでおり、「コンピュータとはプログラムによって指示された通りに計算処理を行う機械である」と考えることができます。
電卓は一度の操作でひとつの計算をして結果を表示することしか出来ません。そこにはプログラムと、途中の結果を残して、前後の計算をむすびつける記憶装置がないからです(*3)。

*1 ここで典型的、と言っているのは「いわゆるパソコンなど、一般によく目にするコンピュータは」という意味です。コンピュータには多くの形式（スタイル）があり、今回紹介しているタイプとは異なるスタイルのものもあります。
*2 「コンピュータ基礎 1・2」クラスで行います。
*3 古典的な、例えば 1970 年代の電卓はそう考えて良いでしょう。しかしいまの電卓はおそらく「ボタンとディスプレイを使って四則演算を行うプログラム」が埋め込まれたコンピュータによって実現されているでしょう。こうした内部のプログラムを置き換えたり変更することができない専用機としてのコンピュータが今はかなり増えています。ハードディスク内蔵ビデオレコーダーなども中身はコンピュータです。

□ プログラミングについて

プログラミングとは、このプログラムを作る作業のことです。つまり一連の作業を手順の列として表現すれば（書けば）いいのですが、そこには厳しい制限があります。コンピュータが実行できる程度の簡単な指示を、コンピュータに理解できる形式にする必要があります。

たとえば「 1 から 10 までの数を足した値を計算する」には以下のように計算手順を指示します。

```
int count,total;
total=0;
for(count = 1; count <= 10; count++) {
  total = total + count;
}
```

これは C 言語によるプログラムの例（部分）です。「 1 から 10 までの数を足した値を計算する」と言う作業を C 言語に翻訳したものと考えれば良いでしょう。「C 言語によるプログラミングを学ぶ」ということはつまりこの C 言語という流儀でコンピュータに手順を伝えるときの表現方法（文法）と、作業の行い方（操作法）を学ぶということにあたります。

> たとえばロボットアームを制御するコンピュータを相手にした場合、「この石を取って」といった指示はおそらくできません。「第一関節を 45 度曲げ、第二関節を 30 度曲げ、指を曲げ、指先のセンサーが on になったら指を曲げるのを止める」と、目的の動作をコンピュータに予め用意された個別の動作に分解して手順の列として与えることになります。そしてこの動きを一段階ごとに C 言語の文法で記述するのです。

□ C 言語について

C 言語は現在もっともよく使われるプログラミング言語の一つです。用途を特に限定した設計でなかったこともあって、数値計算、機械制御、ネットワークサービス、ゲーム、グラフィクス、音楽処理など多くの場面で使われています。

C 言語の生まれはかなり古く、1970 年代前半に米国 Bell Laboratory の Dennis M. Ritchie が開発しました。Ken Thompson と Ritchie が 1978 年に C 言語の解説書として “The C Programming Language” を出版し、当時はこの巻末の Reference manual に掲載されていた C 言語の仕様に基づいて多くの人が C 言語の処理系を開発し、利用していました。その後アメリカの標準化協会である ANSI (American National Standards Institute) が C 言語の標準規格化に努力し、1988 年に承認されて ANSI C と呼ばれる標準規格としての C 言語の詳細が確定しました。

□ プログラミング言語について

プログラミング言語は C 以外にも多くのものがあります。

・Basic - 1965 年に開発された初心者向けの言語。簡単な文法規則が特徴。1980 年前後に 8bit CPU を利用したような小型のコンピュータでよく使われた。現在 Microsoft 関連のソフトウェア開発でよく使われている Visual Basic もこの流れをくむが、高機能になったぶん複雑な言語になった。
・Pascal – 1968 年の開発。構造化プログラミング(*4)に適しており、教育用に多く使われる。1980 年代の Macintosh のシステム開発に使われたり、現在も Delphi というアプリケーション開発用の製品があり実用性も高い。

・Lisp – 1950 年代末から 1960 年代にかけて人口知能用の言語として考案された。関数型言語と呼ばれる場合もある。
・Smalltalk – 1970 年代初期から開発され、1980 年代に公開されたオブジェクト指向プログラミングに特化した言語。
・FORTRAN – 1957 年、IBM の大型汎用機向けに科学技術計算用、つまり三角関数など小数点以下の有効精度の桁数が多いことが重要な計算処理に注力して開発された。
・ COBOL – 1960 年頃に開発、公表された事務計算向けの言語。非常に大きな桁数の整数が自然に扱えるなど、事務処理に適した機能を言語仕様に取り込んでいる。

多くのプログラミング言語が 1960 年代に開発され、今も使われ続けています。例えば FORTRAN は現在もスーパーコンピュータを利用するような科学技術計算でよく使われており、COBOL は言語仕様を拡張しながらいまも事務処理を中心に多用されています。
新しい言語もまた多く登場しています。1985 年には C 言語にオブジェクト指向の枠組みを導入した C++ が発表されています。1987 年には Perl の version 1.000 が発表されました。Java は 1996 年の発表です。

プログラミング言語はこのように用途や目的に応じて設計され、成長していくものです。機会があればそれぞれの言語の歴史や、違いを感じてみてください。

> 例えば C 言語も突然現れたわけではなく、その前身には Martin Richards が設計した BCPL があります。Ken Thompson は BCPL に基づいた B 言語の処理系を作り、これによって最初の UNIX システムを 1970 年に開発しました。C 言語は後に Dennis Ritchie が UNIX を書き直すために作り出されたのです。このあたりの流れは UNIX と C 言語の開発史そのものであり、ダイナミックで興味深いものです。調べてみると良いでしょう(*5)。

*4 構造化プログラミング、オブジェクト指向プログラミング、といったことについてはここでは説明しません。
*5 The Development of the C Language, Dennis M. Ritchie, Bell Labs/Lucent Technologies, 1996
http://roguelife.org/~fujita/COOKIES/HISTORY/BTL/chist.html に藤田昭人による邦訳あり

□ UNIX, Linux について

みなさんにとって標準的な作業環境として用意されているのは Linux という名前の OS (Operating System) です。OS についてはここでは説明しません(*6)が、今は単に作業する環境のことだと思って下さい。Windows もひとつの作業環境であり多くの場所で使われていますが、このクラスでは Linux という作業環境を使って実習を進めます。OS （環境）が変わると実際の操作に関して何がどの程度変わるか、ということについては実際に試して感じると良いでしょう。

Linux は UNIX という OS の、ひとつの実装です。つまり UNIX という種類（パターン）の OS の設計スタイル、ある種の規格があり、それに沿って実際に動くシステムが 1970 年以来、数多く作られてきました。Linux は 1992 年前後に Linus Torvals を中心として作られた、そうした UNIX システムのひとつです。

一番最初の UNIX OS が米国 AT&T の Bell Laboratory で開発されて以来、UNIX の権利はさまざまな会社に移っています。2004 年 8 月現在、AT&T 由来の UNIX OS の知的財産権と、その名称の商標権は分離され、それぞれ SCO 社と The Open Group が保持しています。本文では、AT&T が開発した UNIX OS と互換性のある OS 群をまとめてただ UNIX と呼ぶ場合がああります。

*6 「コンピュータ基礎 1・2」クラスで説明します。

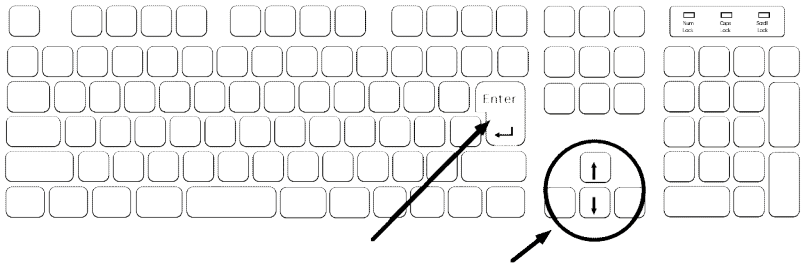
電源投入後のキー入力について

1. 電源投入後

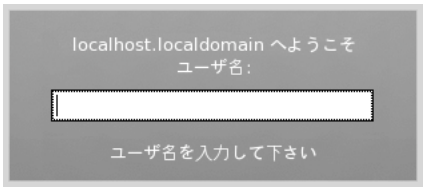
画面上には右のような画像が表示されます。

画面左下の「Vine Linux 3.1」と書かれた部分の白黒反転表示を確認して、Enter キーを押します。もし表示が「WindowsXP」を指していた場合、矢印キー ↑ ↓ で「Vine Linux 3.1」に切り替えて、Enter キーを押します。

矢印キーと Enter キーの位置：

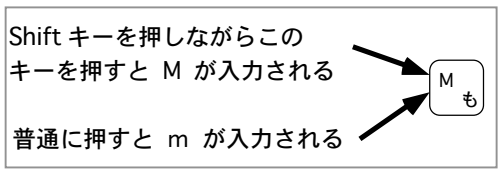


2. ユーザ名とパスワードの入力

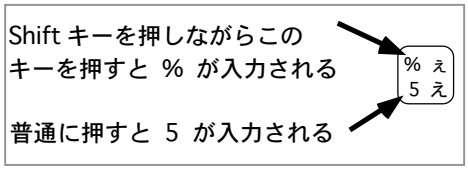


数十秒の時間が必要ですが、さまざまな文字表示が流れていった後に、画面上には右のような表示が出ていると思います。
ここに自分のユーザ名（恐らく g123456 のようなもの）を入力します。まず白く抜けている記入欄にカーソル（｜）があることを確認し、キーボードを一文字ずつ押してユーザ名を入力します。カーソルがない場合は入力できません。マウスポインタ（↖）を記入欄に移動させ、記入欄の上でクリックしてください。打ち間違いは「Back Space」キーで訂正できます。入力できれば Enter キーを押して下さい。
なお、今の段階では数字の入力にはキーボード右側のテンキーは使いません。キーボードの左半分、上から二列目の数字キーを使ってください。
同様の操作でパスワードも入力します。上の作業で Enter キーを押すと、次にパスワードを入力するように表示が変わります。そこで記入欄に今度はパスワード（2dWL#34のようなもの）を入力します。

文字、記号の入力方法：
アルファベットとひらがなしか書かれていないキー、例えば M も は、普通に押すと小文字の「m」が入力されます。Shift キーを押しながら入力すると「M」と大文字になります。

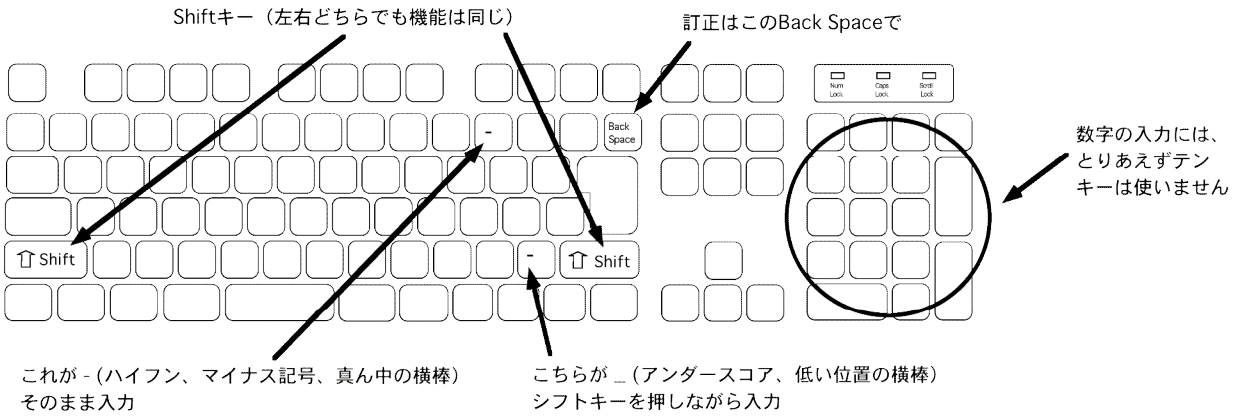


数字が0～9まで横一列に書かれたキー、例えば % え 5 え は、普通に押すと数字の「5」、Shift キーを押しながら入力すると記号である「%」が入力されます。

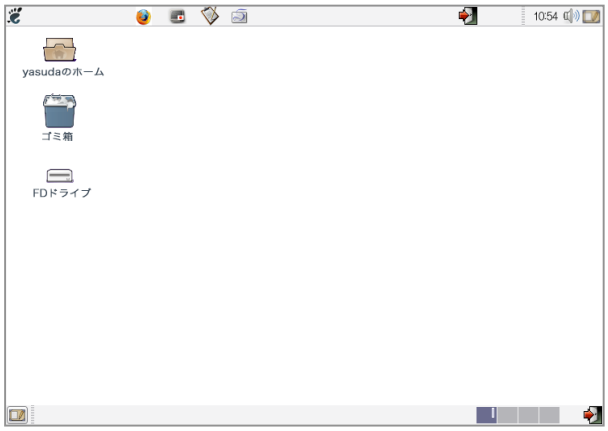


パスワードは、他の人に見られないように、どのような文字を入力しても文字は表示されません。もし正しい文字が打ち込めているか自信がない場合は、いったんユーザ名の記入欄にその記号などを入力してみると良いでしょう。確認できたら「Back Space」キーで消しましょう。

各キーの場所：間違いやすい記号と、Shift キー、Backspaceキーなど



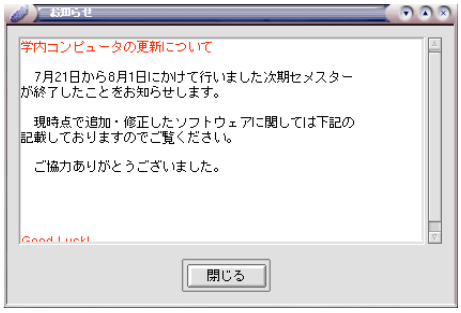
パスワードが入力できたら Enter キーを押してください。
これで Login できるはずです。



Login に成功すると、右図のような画面になっているはずです。

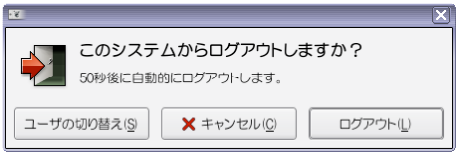
もしそうはならずに、元の画面に戻ったひとは、ユーザ名、パスワードのいずれかを入力し間違えています。もう一度試して下さい。
どうしてもうまくいかない場合は何かを間違えて覚えているか、操作を間違えている可能性がありますので、担当講師（または補助員）に確認してください。

Login 直後の画面に、右図のようなウィンドウが表示されている場合があります。利用者に対するアナウンスが書かれていますので、しっかり読んで、理解するなりメモするなりしてから「閉じる」ボタンをクリックして画面上から消してください。

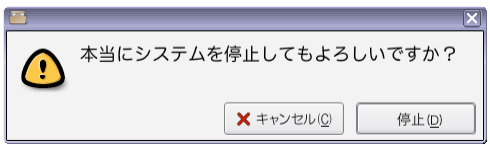


3. Logout と電源断

作業を終了して電源を切るには、画面右上、または右下隅にある  アイコンをクリックしてください。右図のウィンドウが表示されますので、「ログアウト」ボタンをクリックします。
これで Login 前の状態に戻ります。



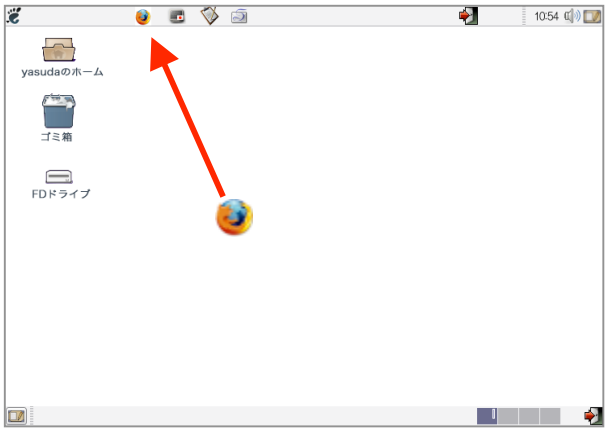
そこで再び画面右下にある  アイコン（電源OFFと書かれています）をクリックすれば、もう一段階確認のためのウィンドウが表示されるので、「停止」ボタンをクリックして下さい。。



これで電源が自動的に切れるところまで進みます。
画面表示が暗くなり、本体のランプが消えたら正常に終了することができています。

授業準備

PCの電源を入れ、login をしたあと、授業を始める前に、まず以下の状態にまでしておいてください。これが授業準備となります。



1. ブラウザを起動してドキュメントを読む

login 後の画面はおそらく右図のようになっているでしょう。まず Web ブラウザを起動します。それには、画面左上のメニュー（パネル）にある、FireFox ブラウザのアイコン をクリックしてください。

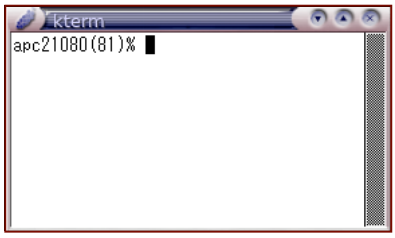
何十秒か待つ必要がありますが、ウィンドウが一つ開いて、右図のような画面になるでしょう。
もし右のページ（学内向けトップページ）を表示させたければ、開いた FireFoxウィンドウの左上のほうにある、ホームボタン をクリックすると良いでしょう。

次に、このクラスの教材ページを表示させます。
開いたウィンドウ、上の方にある、

の領域をクリックしてカーソル（この場合は縦棒）を表示させ、
http://ylb.jp/c/
と入力して Enter キーを押して下さい。表示されたページのなかにこのクラスのリンクが表示されると思います。
そこに教材が置いてありますので、実習時間中はここを常に参照すると良いでしょう。

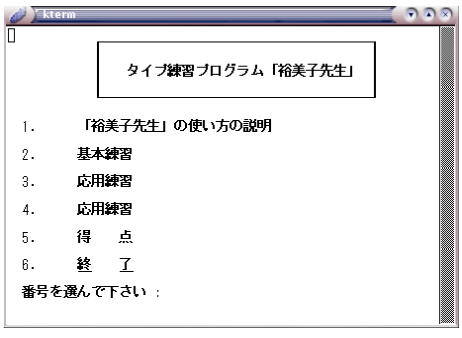
2. ターミナルを起動してコマンド入力状態にする

続いてターミナル（GNOME端末というもの)を起動します。画面左上、FireFox ブラウザアイコン のすぐとなりにある、ktermのアイコン をクリックしてください。右図のように、コマンド入力可能なターミナル画面が現れるでしょう。



3. タイプ練習をして待つ

ターミナル画面で
yumiko
と入力して Enter キーを押すと、タイプ練習が可能な状態になります。クラスが始まるまではタイプ練習をしておくと良いでしょう。



プログラミング演習 A 教材 (#2)

■ GUI と CUI

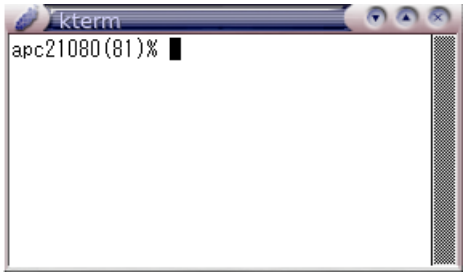
Windows や Macintosh はコンピュータの中にある機能を形にして見せるようにしています。見えるものを操作すれば良いように作られているわけで、これを GUI - グラフィカルユーザインタフェイスと呼びます。

UNIX でも GUI が幾らか用意されていますが、操作のあちこちにコマンドがでてきます。文字によるコマンドとメッセージを中心にした操作系を CUI - キャラクタユーザインタフェイスと呼びます。

ところで CUI の場合、機能の名前を知らなければどんなに便利な機能がコンピュータの中にあったとしても利用できません。名前を間違えて指定すれば、望みの機能とは違う機能が働き出しますので注意が必要です。
GUI では、機能はそれらしく目立つように表現され、メニューとして表示されるのでそういった問題はありません。
UNIX はこの CUI について極めてうまく作り込まれたシステムであり、UNIX を使うには今でも CUI が便利な部分が多くあります。そこでこのクラスでも CUI を中心に実習を行ないます。

■ コマンドとファイル

□ コマンドの実行



ターミナル (kterm というもの)を起動してください。画面左上、Mozilla ブラウザアイコンのすぐとなりにある、kterm のアイコン  をクリック、です。右図のような画面になります。

このターミナルのウィンドウの中に見えている libxt131(81)% といった文字が、コンピュータから最初に提示されているメッセージとなります。このメッセージはプロンプトと呼ばれ、「次のコマンドをどうぞ」という意味です。ここでプロンプトに続けてコマンドをキー入力することで操作を行うわけです。

プロンプトに続いて、下記のコマンドを試して下さい。(以降は UNIX ガイドを参照しながら進めると良いでしょう)
date コマンドで現在の時刻を表示させる <<ガイド 3.3>>
cal コマンドで今月のカレンダーを表示 <<ガイド 3.3>>
who コマンドで login したユーザ名を表示

□ ウィンドウシステム上で動くコマンド

文字を表示するだけでなく、GUI 上で動くコマンドもあります。
xeyes コマンドで目玉を表示します。ウィンドウサイズを変えてどう表示されるか試して下さい。
ウィンドウ右上の  の×印をクリックして終了するか、ターミナルのウィンドウをクリックしてアクティブ状態にして C-c (Control キーを押しながら C キー)で中断します。
この間、プロンプトが表示されず、次のコマンドを与えられなくなっている事を確認してください。

□ ファイルの操作

これから C のプログラムを書いていきます。プログラムは Emacs で書き、ファイルに保存します。Emacs はテキストエディタと呼ばれるソフトウェアで、プログラムなどの文字を書くためのツールです。

ターミナルで emacs & とタイプすると、Emacs が起動し、画面上に Emacs のウィンドウが広がるでしょう。

まず文字入力や Emacs の操作そのものに慣れるために、右図のような簡単な自己紹介を入力してください。以下の手順をまもって作業してください。

自己紹介
理学部コンピュータ科学科
473088 榎田裕一郎

コンピュータ利用歴：
高校一年生から学校の授業で少しずつ利用。
三年生になって自分の PC を購入。
プログラミング経験：
高校二年生のころに授業で少しやったが全く記憶になし。

1. Emacs を起動

2. C-x C-f （Control キーを押しながら x キーを押し、次に Control キーを押しながら f キーを押す）して下さい。Emacs ウィンドウの一番下の行が、下図のような画面表示になると思います。

3. そこで sample.txt とファイル名を入力し、Enter キーを押して下さい。ウィンドウの内容が真っ白になり、画面左下には下図のように sample.txt と (New file) といった表示がされているでしょう。

4. 自己紹介文を入力していってください。かな漢字変換の方法はガイドの <<10.1>> 参照。この、文字の入力、修正作業のことを「編集」と呼んでいます。「Emacs でプログラムを編集する」という具合です。

5. 入力の途中で何度か、現在の状態を保存する（sample.txt ファイルに書き込む）ために C-x C-s（Control キーを押しながら x キーを押し、次に Control キーを押しながら s キーを押す）して下さい。下図のように保存された、という表示がされます。

5. できあがれば最後にもう一度 C-x C-s で保存をします。
最後に保存をしてから全く変更を加えていない場合は C-x C-s しても「No changes need to be saved」と表示され、上のように「Wrote …」とは表示されませんので注意してください。

6. C-x C-c で Emacs を終了します。
　常に終了する必要はありません。次のファイルを編集するときは 2. の C-x C-f からはじめてください。

一般的な Emacs での作業手順はこのようなものです。

念のために、ファイルの中身が正しくできたことを確認しましょう。
まず ls コマンド<<ガイド 5.2>> でファイルの一覧に sample.txt があることを確認し、cat sample.txt として自己紹介文が記録されていることを確認してください。

□ 各種コマンド

ファイルを扱うための主要なコマンドは以下の通りです。(*1)

ファイルの一覧を表示：ls コマンド <<ガイド 5.2>>
ファイルの中身を確認：cat コマンドにファイル名を引数として渡して下さい <<5.4>>
ファイルの複製を作る：cp コマンドに複製もと、複製先のファイル名を指定 <<5.5>>
ファイルを消去する：rm コマンドにファイル名を指定 <<5.5>>

(*1) ここでは説明しません。UNIX ガイドを参照するか、「コンピュータ・リテラシ」クラスでより詳しい説明を聞いて下さい。

プログラミング演習 A 教材 (#2)

■ プログラムの実行

一つプログラムを作って実行してみましょう。

□ 画面に文字を表示する（だけの）プログラム

Emacs を起動して、下のプログラムを入力してください。ファイル名は任意の名前で結構ですが、例えば hello.c とでもしておいてください。C 言語では大文字と小文字（ A と a など）は区別されますので注意してください。

```
#include <stdio.h>

main(){

  printf("My name is Enokida Yuuichiro!");

}
```

入力ができれば、まず保存をしてください。正しく保存できたかどうか、ls コマンドで調べ、cat コマンドで中身を確認してください。
次に cc コマンドでコンパイルします（このコンパイルという作業の意味は後述）。

```
cc2004(86)% cc hello.c
cc2004(87)%
```

その後 ls すると、a.out というファイルが増えていることがわかるでしょう。これを ./a.out として実行して下さい。画面に名前が表示されていくはずです。（以下、作業例。下線が入っている部分は自分でタイプしたところ。）

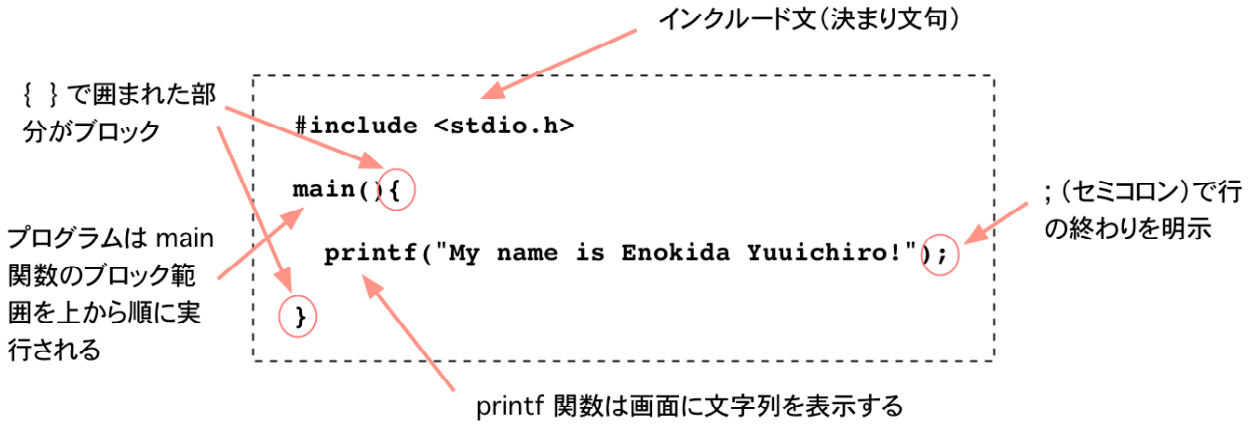
```
cc2004(86)% ls
Mail  Wnn6  public_html   hello.c   sample.txt
cc2004(87)% cat hello.c
#include <stdio.h>

main(){

  printf("My name is Enokida Yuuichiro!");

}
cc2004(89)% cc hello.c
cc2004(90)% ls
Mail  Wnn6  a.out     public_html   hello.c   sample.txt
cc2004(91)% ./a.out
My name is Enokida Yuuichiro! cc2004(92)%
```

それができたら、今度は Emacs で ”My name is …” の部分を自分の名前に変更して保存し直し、再び cc コマンドによってコンパイルし、./a.out によって実行して、変更が反映されていることを確認してください。

これが「C 言語でプログラムを作成し、実行する（そして修正してまた実行する）」という一連の作業の最も簡単な形になります。

□ プログラムの解説

プログラムの内容を行ごとに説明します。



- 1 行目の #include は今は決まり文句として覚えておいて下さい(*1)。
- { } によって囲まれた範囲をブロックと呼んでいます。
- 3 行目の main() 直後の { でブロックが始まり、最下行の } でブロックが終わっており、これが main() 関数(*2)の範囲を示しています。
- 5 行目の printf() 関数は画面に文字を表示する機能を提供します。
- 5 行目の最後には ；（セミコロン）があり、これが printf() 関数の行の終わりを示しています。
- プログラムは main() 関数の中身を上から順に一行ずつ実行します。

(*1) 「プログラミング C 」クラスで学びます。
(*2) 関数については何週か後で説明します。

□ わかりやすいプログラムの表記

2, 4,6 行目の空白や、5 行目の p rintf() 関数の前の空白は、なくても構いません。より見やすくなると思い、入れているものです。これらをすべて詰めてもプログラムとしては正しく機能します。つまり、上のプログラムはこのように書くこともできます。

```
#include <stdio.h> main(){printf(
"My name is Enokida Yuuichiro!");}
```

ただ、長いプログラムでこのような書き方をすると、プログラムが読みにくくなってしまいます。上に示したようにプログラムの中の文には構造があり、記号にも役割があるのですから、それらがよりはっきり分かりやすくなるよう、表記について努力すべきです。（別稿「コーディングスタイル」も参照のこと。）

□ 画面への出力を制御する

5 行目の printf() 関数と、その実行結果である画面上の表示に注目してください。

プログラム：

```
  printf("My name is Enokida Yuuichiro!");
```

実行結果：

```
My name is Enokida Yuuichiro!cc2004(92)%
```

プログラムによって出力された（画面上に表示された）”My name is …” の行のすぐ右にコマンドプロンプト（例では「cc2004(92)%」 ）が表示されて格好が良くありません。（通例、プロンプトは行の左端に表示されるものなので。）

これは printf() の出力指定に「表示したあとで改行する」という指示が含まれていないからです。改行表示がないと、出力は次々に右につながって表示されつづけます。

課題：
以下のような printf() の書き方をするとどのような出力結果になるか試せ
（各行の終わりを示す; を忘れないように注意。）

```
printf("My name ");
printf("is");
printf("Enokida Yuuichiro!");
```

C 言語での改行指定は、表示する文字列に「 \n 」（バックスラッシュと n ）を含めることで行います。具体的には以下のように書きます。

```
printf("My name is Enokida Yuuichiro!\n");
```

つまり「 \n 」は「改行するための文字(*3)」であり、これを画面上に出力すると、その次に出力された文字は次の行の左端から表示がはじまります。

(*3) 実際に C コンパイラは \n を \ と n という二文字としては扱わず、一文字（ASCII コードで 0a (10 番目)の文字）として扱っています。

□制御文字・エスケープシーケンス

こうした \ による特別な文字の表現は他にもあります。

\n -- 改行
\t -- タブコードを表示する
\\ -- \ を表示する
\' -- ' （シングルクォーテーション）を表示する
\" -- " （ダブルクォーテーション）を表示する

など。

課題：
以下のような実行結果になるよう printf() の記述を変更し、実行せよ

```
cc2004(97)% ./a.out
My name is
Enokida Yuuichiro!
I like "Sushi" much.
cc2004(98)%
```

■ コーディングスタイル

プログラムは、コンピュータのためだけにあるのではなく、人間が読んで、その内容を理解するためにもあります。
プログラムを書いた人が読むだけではなく、他の人が読む場合もあります。したがって、読みやすいように、内容を理解しやすいように書く必要があります。読みやすいプログラムを書けるようになるためには、誰か他の人の（できれば上級者の）書いたプログラムのスタイル（コーディングスタイル）を真似て書くことが大事です。

読みやすいコーディングスタイルのための、いくつかのポイントを挙げておきます。

1. 適切な場所に空白を入れる

コンマの後、for(...; ...; ...) のセミコロンの後

```
int x, y, z;
```

代入演算子(=, += など)、比較演算子(==, < など)、二項演算子(+, -, && など)の前後

```
a = x + y;
```

if, for, while, switch などに続く () の前後

```
for (i = 0; i < 10; i++) {
```

2. 適切な場所に空行を入れる。

関数定義と関数定義との間

```
int f()
{
     ...
}

int g()
{
     ...
}
```

関数内の変数宣言文と実行文との間

```
int f()
{
     int a, b;

     a =10;
     ...
}
```

3. 段下げ（インデンテーション）をきちんと行う
　{ } の中身（ブロック）は一段下げる。

```
for (i = 0; i < 10; i++) {
    if (i < 5) {
        a = i;
        while (a > 0) {
            a -= 2;
            if (a == 5) {
                b++;
            } else {
                b--;
            }
        }
    } else {
         switch (i) {
         case 5 :
              b = a;
              break;
         case 6 :
              b = a +1;
              break;
         default:
              b = a +2;
        }
    }
}
```

コーディングスタイルには色々な流儀があり、その中のどれが良いかということは言えません。
しかし、コーディングスタイルに気を遣う、ということと、コーディングスタイルを統一する（少なくとも一つのプログラム中では)、ということは守ってください。

プログラミング演習 A 教材 (#3)

■ コンパイル

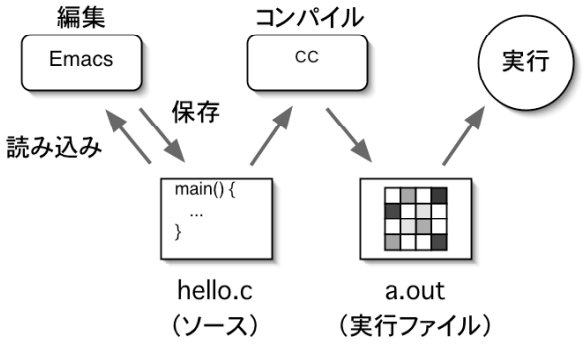
C プログラムはコンパイルという処理を経ないとコンピュータに実行させることができません。

□ コンパイルという変換過程

多くのコンピュータ・システムは、人間が書いたプログラムを直接実行することができません(*1)。C プログラムをコンピュータに実行させるためには、それをコンピュータが実行可能な形式に変換してから実行するというステップを踏みます。この変換のことをコンパイルと呼んでいます。

右図はそれを模式的に書いたものです。

1. たとえば hello.c という C プログラムを Emacs で編集、つまりディスクから読み込んで修正し、ディスクに保存し直していたとします。
2. プログラムができあがれば、それを Emacs で保存し、
3. できあがった hello.c ファイルをコンパイルして a.out というファイルに変換します。
   % cc hello.c とします。
4. できあがった a.out ファイルを、./a.out （先頭にピリオドとスラッシュをつける）として実行します。



こうした関係にあるとき、hello.c をソース（またはソースプログラム）、a.out を実行ファイル（またはオブジェクトプログラム）と呼びます。

*1 多くのコンピュータ・システムは、人間が書いたプログラムを直接実行することができません。ただこれは単にコンピュータが直接実行できるプログラムを人間が書くのは面倒なので、書きやすいプログラミング言語でプログラムを書き、それをハードウェアが直接実行できる言語に機械変換するという方法を多くのケースで採っているというだけのことです。つまり便利さの問題です。原理的には人間が書けるプログラムを直接実行できるハードウェアが作れないわけでも、ハードウェアが直接実行できるプログラムを人間が書けないわけでもありません。

□ コンパイル・エラー

コンパイルは常に成功するとは限りません。例えばプログラムに文法上の誤りがあった場合、コンパイルは中断され、エラーメッセージが表示されます。このとき a.out 実行ファイルは作成されません。

例えば文法上のエラーのあるプログラムをコンパイルした例を示します。(上がソース、下がエラーメッセージ)

```
#include <stdio.h>
main(){
  pritntf("Hello World!")
}
```

```
hello.c: In function `main':
  ★ hello.c の ‘main’ 関数の中で、
hello.c:3: parse error before `}'
  ★ 3 行め：`}'の前が解釈できない
```

ソースの誤りは 3 行目の「printf( )」の最後に「；」（セミコロン）を付け忘れたことです。エラーメッセージを読むと、そのことが簡単な英語で書かれています。実際には表示されませんが、今回の例では簡単な和訳を★に続けてつけてみました。

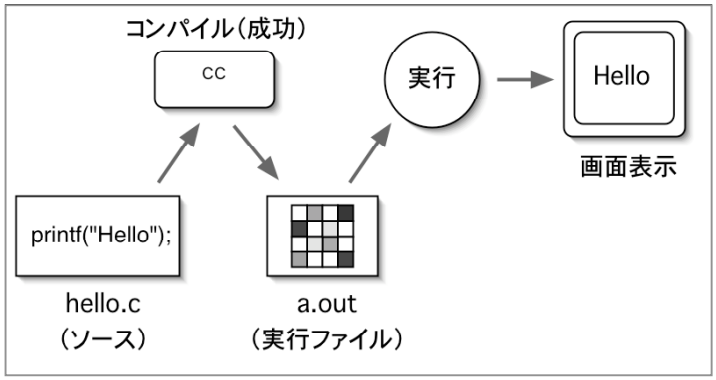
□ コンパイルに失敗した場合の a.out

このようにコンパイルに失敗した場合、a.out 実行ファイルは作成されませんが、削除もされません。つまりはじめてコンパイルした時に失敗しても a.out は作られませんが、以前にコンパイルに成功して a.out が一旦作られていた場合、最後のコンパイルが失敗しても a.out は削除されず、以前に成功してできたものが残っています。

コンパイルエラーの有無をよく見ないでプログラムを修正していると、エラーを起こしているにもかかわらず、たまたま古い a.out 実行ファイルを実行し、プログラムは修正した筈なのに結果が変わっていない、何故なんだろう？というように誤解する場合があります。

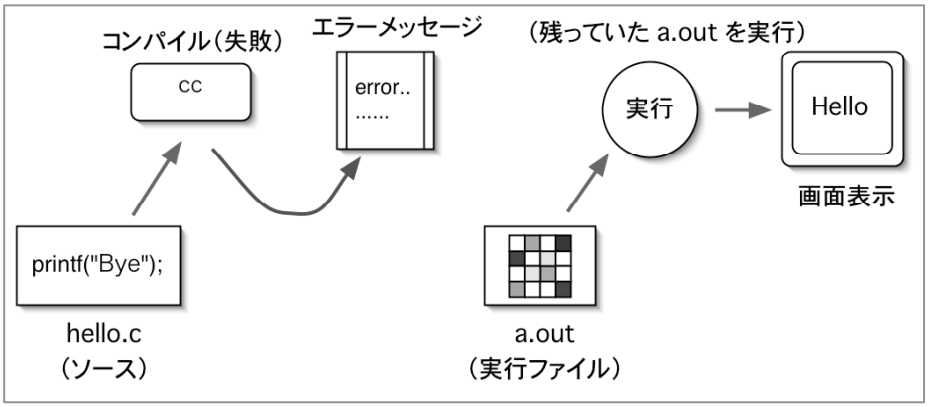
例えば Hello と画面に表示するプログラムを作っていたとします。

うまくできてコンパイルに成功し、実行して画面表示を確認しました。(右図)



このあと、Hello ではなく Bye と画面に表示するようにプログラムを修正しました。ところが修正を失敗して、コンパイルエラーが出てしまいました。(右図)

このとき a.out は書き換えられていません。たまたま前回作成された a.out が残っているだけです。



これではいくら実行しても画面表示は Hello のままです。何度 Hello を Hi や Yahoo に直してコンパイルし直しても、実行結果（画面表示）は変わりません。まずコンパイルエラーの原因を直す必要があります。

□ ワーニング

はじめのうちは遭遇しないと思いますが、重大な誤りではないため、warning （警告）だけで実行ファイルの作成まではする、というエラーもあります。以下のように先頭に warning と出ます。

```
sample.c:7: warning: assignment makes integer from pointer without a cast
```

理解して欲しいこと：
コンパイルエラーは必ずその原因を直して下さい。warning はともかく、それ以外の全てのコンパイルエラーを修正してからでないと、プログラムは一行も実行できないのです。

□ エラーメッセージの例

・引用符を閉じ忘れた場合のエラーメッセージ

printf( ) 関数の中には表示したい文字列を引用符で囲って指定します。その引用符を閉じ忘れた場合は、以下のようなエラーになります。

```
#include <stdio.h>
main(){
  pritntf("Hello World!);
}
```

```
hello.c:3: unterminated string or character constant
  ★ hello.c: 3 行目 : 終端処理がされていない文字列または文字定数あり
hello.c:3: possible real start of unterminated constant
  ★ hello.c: 3 行目 : 終端処理されていない定数が実際に開始されたのはこの行の可能性あり
```

行数（ 3 行目）を頼りに、何か文字列まわりで間違えたことをしていないか捜せば見つかるでしょう。

・関数の名前を間違えた場合のエラーメッセージ

例えば printf( ) 関数の名前を prittf( ) に間違えた時は以下のようなエラーになります。

```
#include <stdio.h>
main(){
  prittf("Hello World!");
}
```

```
/tmp/cc8DjRo7.o: In function `main':
  ★ 一時ファイル /tmp/cc8DjRo7.o で： main 関数の中で、
/tmp/cc8DjRo7.o(.text+0xf): undefined reference to `prittf'
  ★ 一時ファイル /tmp/cc8DjRo7.o で： prittf という未定義のものを参照している
collect2: ld returned 1 exit status
  ★ collect2: ld は 1 を終了ステイタスとして返したぞ（通常はゼロが返る）
```

少々わかりにくいエラーメッセージとなりましたが、関数の名前を間違えるとこのようになります。この場合は行数が表示されていませんので、prittf という名前をてがかりに捜すことになるでしょう。

経験則：
エラーメッセージは注意深く読む。
それでもエラーの場所が分からないときは最後に修正したところの周辺を疑うべき。

・複雑なケース

（この例は最初はあまり気にしないで下さい。多少なりとも複雑なプログラムを書くようになったら遭遇するケースとしていつかまた見直して下さい。）

以下のようなプログラム（ 1 から 10 までの合計を表示する）があります。

```
#include <stdio.h>
int main() {
  int count,total;
  total=0;
  for(count = 1; count <= 10; count++) {
    total = total + 1;
  }
  printf("result=%d\n", total);
  return 0;
}
```

for() のブロック

main() のブロック

このプログラムの 7 行目の }; を何かの拍子に間違えて消してしまったとしましょう。そのとき、エラーメッセージはこのようになります。

```
total.c: In function `main':
  ★ total.c: の main 関数のなかで、
total.c:11: parse error at end of input
  ★ total.c: 11 行目:入力の最後として解釈できない部分あり（入力の最後とは思えない）
```

7 行目にエラーがあるはずなのに 11 行目、つまり文末（入力の最後）にエラーがある、と言われてしまいました。11 行目をいくら見てもこれではエラーの原因がわかりません。

これは 5 行目の for( ) ではじまった { } で囲まれたブロックが、本来 7 行目で終わるはずだったのに、それがなかったため 11 行目まで続いていると解釈された結果です。
ところがこのプログラム全体は 2 行目の main() ではじまった { によるブロックで囲われているはずだったのに、11 行目は 5 行目の for( ) の { とペアだと解釈されたため、その後ろにまだ 2 行目の main( ) の { とペアの } があるに違いない、とコンパイラは考えました。
しかし何もないので「11 行目まで来たが、これが最後とは思えない」というエラーが出てくるわけです。
「分からないときは最後に修正したところの周辺が怪しい」という鉄則をおぼえてください。

```
#include <stdio.h>
int main() {
  int count,total;
  total=0;
  for(count = 1; count <= 10; count++) {
    total = total + 1;

  printf("result=%d\n", total);
  return 0;
}
```

}; を消してしまった！

for() のブロックはここまでだな？

main() のブロックの終わりがない！

プログラミング演習 A 教材 (#3)

■ 計算式、変数と代入

□ 簡単な計算を含むプログラム

Emacs を起動して、下のプログラムを入力、コンパイル、実行してください。時、分、秒をもとに、0 時から数えた秒数を計算して表示するものです。
サンプルでは 10 時 32 分 45 秒としていますが、時間は適当に現在時刻を入れて下さい。ファイル名は任意の名前で結構ですが、例えば comp.c とでもしておいてください。

```
#include <stdio.h>

/*
  秒数を計算する 473088 榎田裕一郎
*/

main() {

  printf("今は %d 時 %d 分 %d 秒です\n", 10, 32, 45);
  printf("全部で %d 秒めですね\n", 10 * 3600 + 32 * 60 + 45);

  return 0;
}
```

押さえて欲しいポイント：
・/* と */ で囲まれた範囲はコメントです。
（コンパイル時には無視されるので、プログラム以外の覚え書きなどを書いておくのに便利。）
・C プログラムの中では数値や計算式が書ける。
・数値を printf で表示させるためには %d といった変換文字列を使う。
・return でプログラムの実行を終了する。返り値（ステイタス）は 0 （正常終了を意味する）。

□ 計算式、演算子、優先順位

C プログラムのなかでは各種の計算が記述できます。四則演算の表記法は以下の通りです。
+, * といった記号を演算子と呼んでいます。

演算子	意味	表記例と計算結果
+	加算（＋）	20 + 6 （結果は 26 になる）
-	減算（－）	20 - 6 （結果 は 14 になる）
*	乗算（×）	20 * 6 （結果 は 120 になる）
/	除算（÷）	20 / 6 （結果 は 3 になる）
%	剰余（割った余り）	20 % 6 （結果 は 2 になる）

整数どうしの割り算 (/) の結果は（小数点以下を切り捨てて）整数となります。
例えば 6 / 20 は 0 です。

通常の計算式と同じく、演算子には優先順位があります。*, /, % が +, - より高い優先順位で処理されます。また、カッコを使った演算順序の制御もできます。また、例では読みやすくするために適度に空白を入れていますが、詰めて書くことも可能です。
つまり以下の計算式はどれも同一の結果を返します。

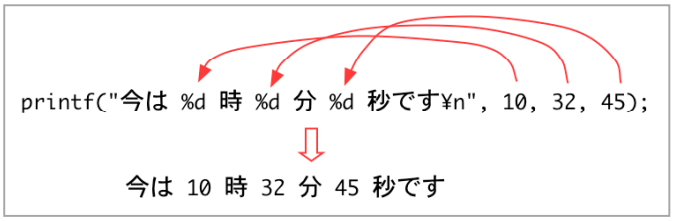
```
10 * 3600 + 32 * 60 + 45
( 10 * 3600 ) + ( 32 * 60 ) + 45
( ( 10 * 3600 ) + ( 32 * 60 ) + 45 )
( ( ( 10 ) * 3600 ) + ( 32 * 60 ) ) + 45
(10*3600)+(32*60)+45
```

□ printf　の変換文字列

printf( ) のカッコのなかにはカンマで区切られた幾つかの項目があります。これらは引数（実引数または argument (*1) ）と呼ばれ、そこには数値 (10, 32 など)、文字列 ("Hello" など)、計算式 (10 * 3600)などが書けます。

最初の引数には文字列を与えます。printf はこの文字列の記述に従って出力結果を整形します。たとえば「今は %d 時」の %d は「ここに 10 進数(decimal) の数値をあてはめる」という指定で、第二の引数である 10 が対応します。こういった % に続く書式（フォーマット）の指定を行う表現のことを「変換文字列」と呼んでいます。

続く二番目の %d には第三引数である 32 が、三番目の %d には第四引数の 45 が対応し、最終的には「今は 10 時 32 分 45 秒です」と整形されて出力（画面に表示）されます。



*1 仮引数(parameter)、実引数(argument) の詳しい用語の定義などについては「プログラミング B」クラスで説明します。

注意：数値と文字列のなかの数字
C 言語のなかでは文字列としての数字と、値として扱われる数値はまったく別のものです。
1 + 2 は数値による計算式で、最終的に 3 という数値になって処理されます。
"1 + 2" は数字を含む文字列であり、数値になることはありません。
つまり、

```
printf(“12+34\n”);
printf(“%d\n”, 12+34);
```

は全く違う結果を画面上に表示します。その理由が納得できましたか？

□ 変数と代入

C 言語（とそれ以外の多くのプログラミング言語）には変数というものがあります。以下にサンプルを示します。結果として「合計は 3 です」と表示されることが想像つくでしょう。

```
int a, b, c;

a = 1;
b = 2;
c = a + b;
printf("合計は %d です\n", c);
```

変数とは値を入れる容器のようなもので、C 言語の場合は中に一つだけ値を入れることが出来ます(*1)。この値を入れる作業を代入と呼び、値を入れるには以下のような代入文を用います。

a = 1;

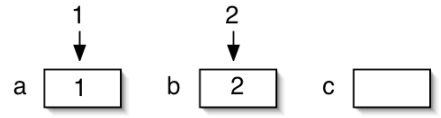
イコールの右辺の値（または計算結果）が左辺の変数に書き込まれます。左辺には一つの変数しか書けません。

ざっと動きを図解すると以下のようになります。

STEP 1.

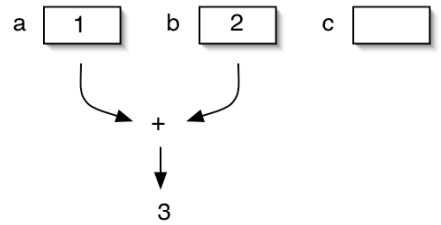
int a, b, c で三つの変数を用意。
C 言語では変数をはじめに宣言します。

STEP 2.



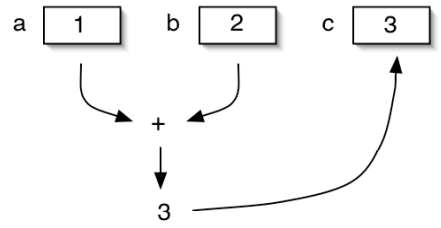
= によって変数 a と b に値を入れる（代入）

STEP 3.



a + b を計算
（ a の中身と b の中身を足す）

STEP 4.



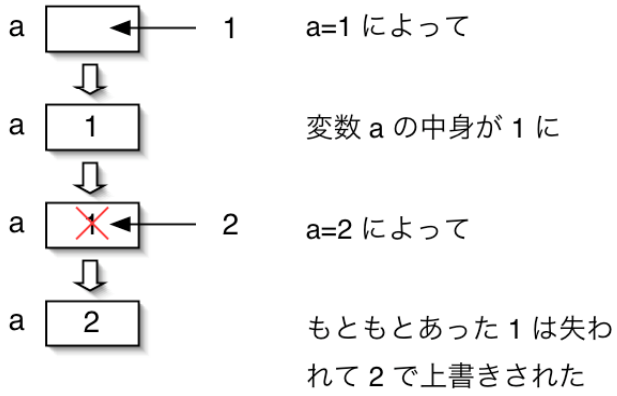
その結果（値）を変数 c に入れる（代入）

*1 配列変数、という手法を使って少し違うこともできます。「プログラミング B」クラスで説明します。

変数には一つの値しか入れられませんから、変数に二度目に値を代入すると以前の値は失われ、それに置き換わって新しい値が残ります。

```
a = 1;
a = 2;
printf("%d\n", a);
```

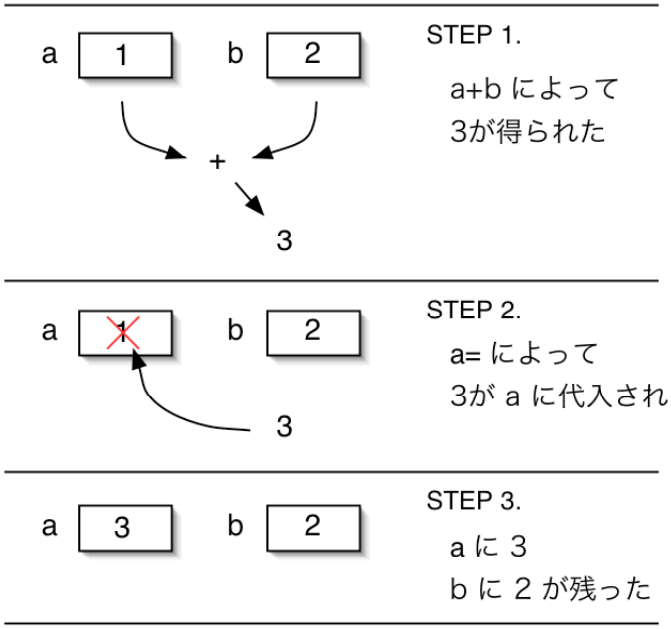
printf( ) の結果は 2 になるでしょう。



計算結果を自分自身に代入することもできます。以下のサンプルの a = a + b に注目して下さい。

```
a = 1;
b = 2;
a = a + b;
printf("%d\n", a);
```

printf( ) の結果は 3 になるでしょう。



重要なこと：
代入文では、右辺の計算が完全に終わってから左辺の変数に代入されます。
右図を見ても分かるように、プログラムには順序と時間の経過が重要なのです。

課題：
冒頭の秒数を計算するプログラムを、変数を使って書き直せ。具体的には `printf()` の記述を以下のように変更し、その前に各変数の値を設定・計算する代入文を加える。

```
printf("今は %d 時 %d 分 %d 秒です\n", hour, min, sec);
printf("全部で %d 秒めですね\n", total);
```

プログラミング演習 A 教材 (#4)　＝これまでに出てきた機能などのまとめ＝

□ 構造、文法

```
#include <stdio.h>
/*
  簡単な計算 473088 榎田裕一郎
*/
main() {
  int a, b, c;
  a=1;
  b=2;
  c=a+b;
  printf("c は %d です。\n",c);
  return 0;
}
```

C 言語のプログラムには構造や意味があります。

- 行はそれぞれ ; で区切られる。
- /* と */ で囲まれるコメント文。
- #include ではじまるインクルード文。
- main 関数。
- { } で囲まれたブロック。

などの要素があります。それぞれ意味があり、プログラムは実行されると main 関数の手続きを上から順に実行します。この決められた記述ルールのことを文法と呼んでいます。

□ 変数と代入文

変数とは値を入れることができる入れ物と考えてください。
「 c=a+b; 」は代入文と言われ、= の左辺の変数に右辺の計算式の結果を入れるものです。
この例では変数 c の値が、変数 a と変数 b の値を加えたものになります。

□ printf と改行

printf( ) 関数は画面上に文字を出力（表示）します。サンプルプログラムの
「 `printf("c は %d です。\n",c);` 」は、画面上に「 c は 3 です 」と表示します。
この「\n」（バックスラッシュと n ）は改行を意味する制御文字です。
%d は、第二引数以降の値をはめこんで出力するための変換文字列です。

□ 演算子、計算式

演算子：　＋ 加算　　－ 減算　　* 乗算　　/ 除算（商）　　% 剰余（除算の余り）
（整数どうしの除算の結果は少数以下が切り落とされて整数になる点に注意。）

計算式には変数、値、演算子、カッコなどを含めることが出来ます。

```
10 * 3600 + 32 * 60 + 45
( a * 3600 ) + ( b * 60 ) + c
( (x+a)*2 + (y+b)*3 ) / c
```

□ コンパイルと実行

% cc hello.c

でコンパイルします。すると a.out という実行ファイルが作られます。(ls コマンドで確認し、実行するには ./ （ピリオドとスラッシュ）を前に付けてファイル名 (a.out) を指定します

% ./a.out

プログラミング演習 A 教材 (#4)

■ 宣言文、データ型

右のサンプルプログラムを入力し、実行して結果がどうなるか調べてください。ファイル名は任意で構いません。

```
#include <stdio.h>

/*
  型を確認する 473088 榎田裕一郎
*/

main() {
  char c;
  int i;
  float f;
  double d;

  c = 'x';
  i = 1234;
  f = 3.14159f;
  d = 1.0e-5;

  printf("char type : %c\n", c);
  printf("int type : %d\n", i);
  printf("float type : %f\n", f);
  printf("double type : %e\n", d);

  return 0;
}
```

□ 宣言文とデータの型

変数を利用するためにはプログラムのはじめに宣言文が必要です。

```
int i;
i=1234;
```

といった形です(*1)。変数やデータ（の値）には「型（かた）」と呼ぶ概念があり、この宣言文では変数を型と共に宣言します。「 int i; 」は、「int 型（がた）」の変数 i を宣言する、という意味です。

変数はその型に適合する値しか扱いません。int 型は整数を扱うための型で、int 型変数 i には整数のみを代入することができます。

同じデータ型の変数なら、「 `int i, j, k;` 」と、カンマで並べて一度に宣言できます。

代表的な型を以下に示します。

char : 文字(character)を扱うための型。`'x', 'a'` など。
int : 整数(integer)を扱うための型。123, 0, -3 など。
float , double: 実数を扱うための型。123.456, 0.0001, 0.0, 1024.0 など。

float と double の違いは精度（有効桁数）です。float の倍の有効桁数を double は持っています。123456.78901234 など。float を単精度浮動小数点型、double を倍精度浮動小数点型と呼ぶ場合があります(*2)。

注意：文字と文字列
C では文字と文字列を全く違うものとして扱っています。文字とは長さが 1 に固定されており、`'x', 'a'` などとシングルクォーテーションで囲んで表現します。文字列とは長さが不定で、`"sample", "hey, Jude"` のように引用符（ダブルクォーテーション）で囲まれています。
つまり `"x"` は「長さが 1 の文字列」であり、`'x'` の「文字 x 」とは異なるものです。
C 言語には文字型はありますが、文字列型は（興味深いことに）用意されておらず、文字の配列として文字列を表現します。配列についての説明は「プログラミング B」で行います。

*1 宣言文無しに変数を使おうとした場合はコンパイルエラーになります。エラーメッセージは：
`sample.c:3: `i' undeclared (first use in this function)`
*2 単精度、倍精度の意味や、具体的な桁数、浮動小数点の意味などについては「＊＊＊」クラスで説明します。

□ 定数における型の明記

整数と実数の型の区別は小数点で行います。 123 は整数、123.456 と小数点をつけた数値は実数とみなされます。123 と 123.0 は数値としては同じですが、型は異なる、というわけです。

123.456 は実数でも double 型とみなされます。float 型と明記したい場合は 123.456f と最後に f をつけて表記します。（サンプルプログラムの `f = 3.14159f;` を参照。）
実数はまた `1.2e-5;` というように表記できます。（ 1.2×10^{-5} つまり `0.000012` ）

□ 型変換、キャスト

int 型変数は整数しか扱えません。そこで int 型変数に実数を代入すると、自動的に型変換が行われ、小数部が切り落とされて整数部だけが代入されます。
つまり `int i; i=123.456;` なら、i には少数以下が切り落とされた 123 が代入されます。

実数変数に整数を代入した場合は、単純に実数化された値が入ります。`float a; a=123;` なら、123 が実数化されて 123.0f となって a に代入されます。

計算式に整数、実数の値や変数が混在していた場合は、精度の高い方に合わせて型変換が行われてから計算が行われます。`float a; a=1.5f * 3;` なら、1.5f * 3 は 1.5f * 3.0f として計算され、4.5f が a に代入されます。

注意が必要なのは /（割り算）で、整数と整数の割り算は整数となって余りは捨てられますが、実数と実数、または実数と整数の割り算では可能な限りの精度で小数点以下まで求められます。

つまり 10 / 4 は 2 ですが、10.0 / 4.0 は 2.5 です。変数を用いた計算でも同じことで、
```
int i, k; float a, b;
i=10; k=4; a=10.0; b=4.0;
```
の場合、
i / k は 2 ですが、a / b は 2.5 です。
i / b や a / k のように実数と整数を混在させた場合は、整数側が実数化されて計算され、結果はともに 2.5 になります。

これらは暗黙の型変換と呼ばれますが、明示的に指示して行うこともできます。

もし、`i / k` の計算を実数化して行い、2.5 という結果を実数変数に代入したい場合は、
`float x; x = (float)i / (float)k;` と書きます。

こうした、値の直前にカッコで囲んで型名を書く「キャスト」と呼ばれる方法で、値の型変換を明示的に指示できます。

キャストの有効範囲がどこまでか不安になるような場合があり得ます。例えば、
`x = (int) a / k * 100;` のようなケースは、
`x = (int)(a) / k * 100;` として a だけキャストされるのか、
`x = (int)(a / k * 100);` のように、全体の計算結果に最後に一度だけキャストされるのか、不安に思うかも知れません。そうしたときは、なるべくはっきりキャストの範囲が明確にわかるよう、上記のようにカッコをうまく使って記述すると良いでしょう。

□ 変数名と予約語

変数は宣言時に名前がつけられますが、変数名として利用可能なのは以下のようなものです。

・最初の一文字は英字（大文字、小文字のいずれでも）ではじまること。
・二文字目以降は英字または数字が使える。
・変数名も大文字と小文字は区別される。（ C の変数は伝統的に小文字で表記される。）
・予約語以外の名前でなければならない。

予約語とは int や float 、return といった、C 言語の文法上、既に使われている単語のことです。以下に予約語の一覧を示します。

```
auto      break     case      char      const     continue  default
do        double    else      enum      extern    float     for
goto      if        int       long      register  return    short
signed    sizeof    static    struct    switch    typedef   union
unsigned  void      volatile  while
```

なお、C 言語では「 _ 」（下線、アンダーライン）は英字とされています。長い変数名をわかりやすくする場合などに使われます。（leftbutton より left_button の方が読みやすい。）

注意：予約語ではないが衝突を避けたい名前
printf( ) などは C 言語の予約語ではなく「一般的な関数」として標準的に UNIX システムに用意されているものです。この名前と衝突するのも良くありませんので、注意しましょう。
こうした標準的なライブラリ関数の名前は多すぎてここには出しません。
変数名を決める際に、同じ名前のライブラリ関数があるかもしれない、と不安になった場合は man コマンドで確認すると良いでしょう。たとえば man 3 printf とすると printf( ) 関数の詳細なマニュアルが表示されるでしょう。

□ データの型に合わせた変換文字列

サンプルプログラムの printf( ) を見てください。文字は %c、整数は %d 、実数は %f または %e と、変換文字列が使い分けられています。

```
printf("char type : %c\n", c);
printf("int type : %d\n", i);
printf("float type : %f\n", f);
printf("double type : %e\n", d);
```

%10d のように、% と変換文字の間に桁数の指定などができます。

代表的な変換文字列

	意味	使用例	その結果
%c	文字を表示	printf("[%c]¥n",'a');	[a] 文字がそのまま表示される。
%d	整数を表示	printf("[%d]¥n",10);	[10] 桁数不定
		printf("[%5d]¥n",10);	[10] 5 桁で表示。不足分は空白。
		printf("[%05d]¥n",10);	[00010] 5 桁。不足はゼロで埋める。
%f	実数を表示	printf("[%f]¥n",12.345);	[12.345000] 桁数不定
		printf("[%9.5f]¥n",12.345);	[12.34500] 小数点含めて全体が 9 桁、小数以下が 5 桁。
%e	実数を表示	printf("[%e]¥n",12.345);	[1.234500e+01] 浮動小数点で表示

プログラミング演習 A 教材 (#5)

■ for によるループ

□ ループ

右のプログラムを入力して実行してください。0 から 9 までの数字を出力するはずです。

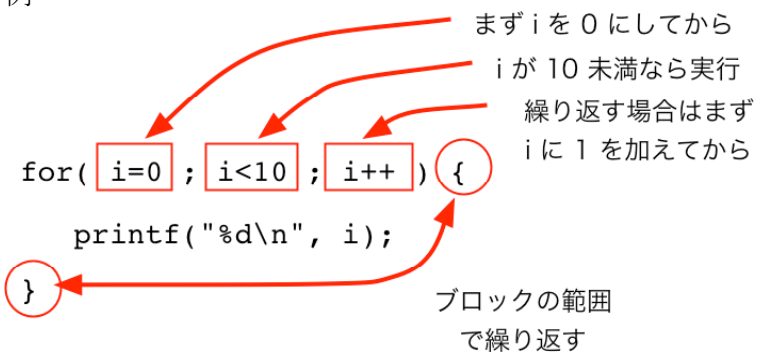
一行しかない printf( ) が 10 回繰り返して実行されています。こうした繰り返し処理のことをループと言います。

```
#include <stdio.h>

/*
  for によるループ 473088 榎田裕一郎
*/

main() {
  int i;

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

□ for 文

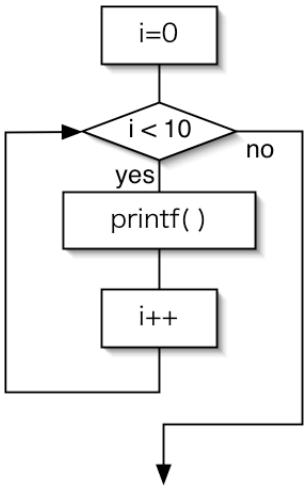
書式：
for(開始時処理 ; 繰り返し条件式 ; 繰り返し毎処理){
　　繰り返す処理
}

例



注意：i++ は「変数 i に 1 を加える」という意味で、「 i = i + 1 」と等価です。（後述）

動作：
for 文は、
・まず開始時処理を実行し、
・その結果が繰り返し条件式に反しておれば何もせず終了
・条件式に当てはまれば続くブロック（ { } の範囲）を実行
・繰り返し毎処理を一度だけ実行してから、
・繰り返し条件判定からくりかえし。
という動作をします。（右図）



プログラムを見ると、for() に続くブロックの処理が 10 回繰り返され、その間に変数 i の値が変化したことがわかると思います。
i が 10 となり、繰り返し条件である i < 10 を満たさなくなった時点で終了しています。

□ 条件式

i < 10 のように、何かの条件を判定する場合に用いるのが条件式です。(条件式についての詳細は if 文の説明で行います。)

演算子	意味
==	等しい
!=	等しくない
>	左辺が大きい
>=	左辺が等しいか大きい
<	左辺が小さい
<=	左辺が等しいか小さい

==, != を等値演算子、>, < などを関係演算子と呼んでいます。
これらより算術演算子のほうが優先度が高いため、
a < b-1 という記述は a < (b-1) と同じとみなされます。
(左から順に処理されてまず a < b が先に処理されるようにはなりません。)

□ 代入演算子

(例にはでていませんが) a=a+10; を a+=10; と書くこともできます。このように変数への代入処理を対象にした演算子を代入演算子と呼びます。

演算子	利用例 (a を 20、b を 6 とする)	同じ意味の記述
+=	a+=b; (a は 26 となる)	a=a+b;
-=	a-=b; (a は 14 となる)	a=a-b;
*=	a*=b; (a は 120 となる)	a=a*b;
/=	a/=b; (a は 3 となる)	a=a/b;
%=	a%=b; (a は 2 となる)	a=a%b;

□ インクリメント、デクリメント

i++ は「変数 i に 1 を加える」という意味で、「 i = i + 1 」「 i += 1 」と等価です。これをインクリメントと呼びます。同様に減算(デクリメント)もあり、こちらは i--と書きます。

演算子	利用例 (a を 20 とする)	同じ意味の記述
++	a++; (a は 21 となる)	a=a+1; や a+=1;
--	a--; (a は 19 となる)	a=a-1; や a-=1;

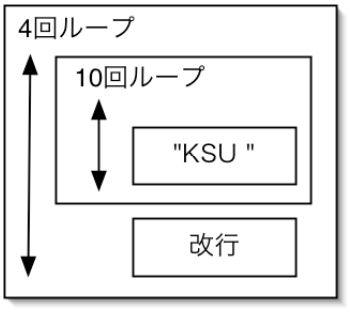
これらは ++a 、--a のように使うこともできます。++a は演算に先立って加算され、a++ は演算の後で加算されます。違いの分かる使用例を以下に示します。

利用例 (a を 20 とする)	実行後の結果
b = 6 * a++;	a は 21、b は 120
b = 6 * ++a;	a は 21、b は 126

「 6 * a++ 」は 6 と掛け合わされたあとで加算され、「 6 * ++a 」は 6 と掛け合わされる前に加算されています。慣れないうちは a++; のように、一行だけの記述で、つまり a=a+1; の代わりに使うのがよいでしょう。

■二重ループ

for( ) 文を二重に重ねることもできます。
右のプログラムを実行して結果を確認してください。
10 回 KSU を出力し、その後改行を出力、これを 4 回繰り返します。




```
#include <stdio.h>

/*
  for による二重ループ 473088 榎田裕一郎
*/

main() {
  int i, j;

  for(i=0; i<4; i++) {
    for(j=0; j<10; j++) {
      printf("KSU ");
    }
    printf("\n");
  }
  return 0;
}
```

□ 課題 1.

一行ごとに KSU の表示が一つずつ増える（10 個まで）プログラムを作成してください。

```
KSU
KSU KSU
KSU KSU KSU
                （中略）
KSU KSU KSU KSU KSU KSU KSU KSU KSU
KSU KSU KSU KSU KSU KSU KSU KSU KSU KSU
```

□ 課題 2.

以下のように右詰めの表示になるようなプログラムを作成してください。

```
 KSU KSU KSU KSU KSU KSU KSU KSU KSU KSU
     KSU KSU KSU KSU KSU KSU KSU KSU KSU
                （中略）
                                KSU KSU  KSU
                                    KSU  KSU
                                         KSU
```

プログラミング演習 A 教材 (#6)

■ for 以外の方法によるループ

□ while

右のプログラムを入力して実行してください。0 から 9 までの数字を出力するはずです。

```
#include <stdio.h>

/*
  while によるループ 473088 榎田裕一郎
*/

main()
{
  int i=0;

  while (i < 10) {
    printf("%3d¥n", i);
    i++;
  }
  return 0;
}
```

while 文はカッコの中の条件式が真である限り、それに続くブロックの処理を繰り返し続けます。

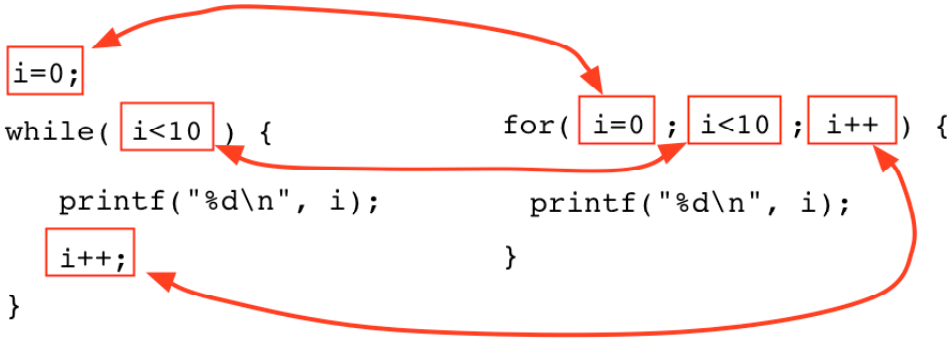
書式：

```
while( 繰り返し条件式 ) {
    繰り返す処理
}
```

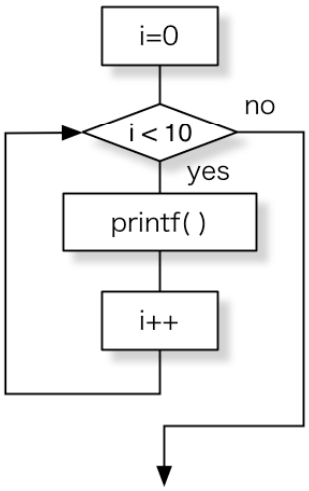
注意：`int i=0;` というのは宣言と同時に 0 を代入するというもので、
`int i;`
`i=0;`
と書いた場合と等価です。

□ for と while の比較

for 文で同様のプログラムを作りましたが、while を用いてこのように書くことも出来ます。for における初期処理、繰り返し条件、繰り返し毎処理が while 文でどのように配置されているかに注意して下さい。



右の図は for による処理の流れを説明したときに用いたものですが、今回の while による記述もまったくこの流れの通りに行われています。つまり例にあげた while と for によるループ処理は全く等価なものです。



あるループを for で書くか、while で書くかはプログラマの判断に任されています。より読みやすく、わかりやすい方法で記述するように書き方を選ぶようにして下さい。（この例のような単純な処理ではどちらを選んでも大差ありませんが、より複雑な処理になるにつれ、こうした選択が重要になってきます。）

□ do ~ while 文による表現

do ~ while 文によるループの表現方法もあります。

右のプログラムを実行して動作を確認してください。

do 文は、それに続くブロックの処理をまず行い、その後に while ( ) に示された条件文による判定を行います。

書式 :

```
do {
    繰り返す処理
} while( 繰り返し条件式 );
```

```
#include <stdio.h>

/*
  do while によるループ 473088 榎田裕一郎
*/

main()
{
  int i=0;

  do {
    printf("%3d\n", i);
    i++;
  } while (i < 10);
  return 0;
}
```

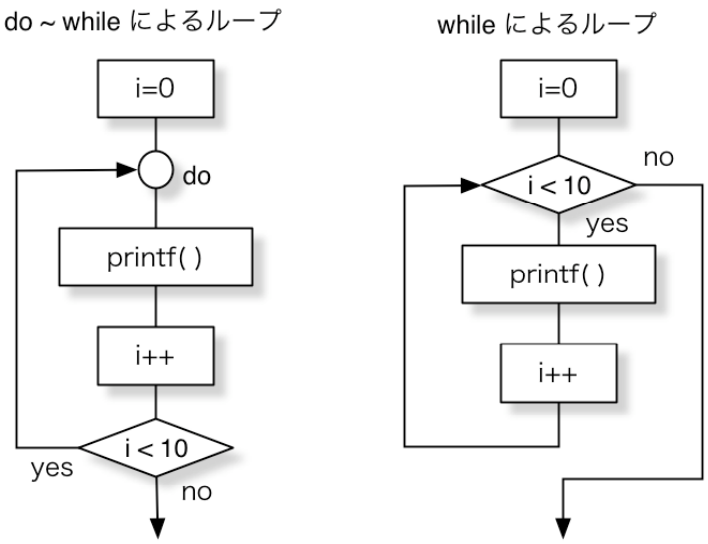
□ do ~ while 文と while 文の比較

両者の処理の流れは右図のようになります。条件判定の位置が異なることに注目して下さい。

注意点：

while ( )文では、括弧内の条件判定が合わないと、繰り返し処理部分を 1 度も実行しないことがある。

do~while ( )文では、最低 1 回は繰り返し処理部分を実行する。



□ 課題 1.

以下のように九九の表を出力するプログラムを while を使った二重ループで作って下さい。

```
 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
```

□ 課題 2.

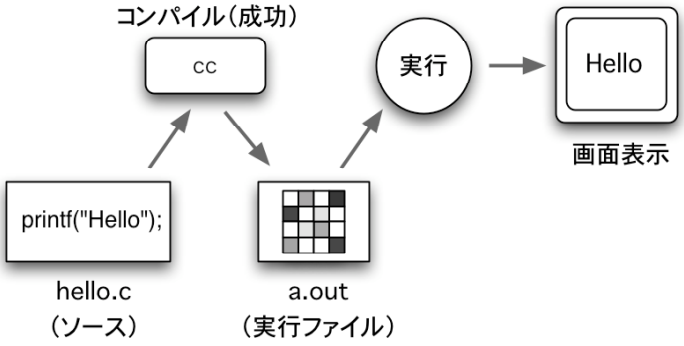
同じく九九の表を、do ~ while を使って作り直して下さい。

■ オブジェクトファイルの指定

□ コンパイル復習

今までプログラムは以下のようにしてコンパイルし、実行していたと思います。

```
cc2000% cc hello.c
cc2000% ./a.out
```

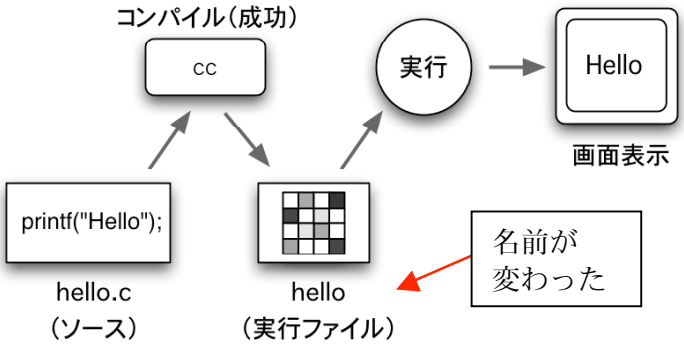


これは hello.c プログラムを cc コマンドによってコンパイルした結果、a.out という名前の実行ファイルが出来たので、それを実行していたわけです。

□ 実行ファイルの名前を指定する

この実行ファイルの名前を以下のようにしてコンパイル時に指定することができます。

```
cc2000% cc -o hello hello.c
cc2000% ./hello
```



cc コマンドの –o オプションに続けて作成する実行ファイル名を指定できます。実行時は作成されたファイル名を指定しますので ./hello となります。

例のようにソース名の拡張子（.c）を外した名前を実行ファイル名にする場合が一般的です。（違う名前を付けても構いません。）

□ 使いみち

そろそろ課題のプログラムなどが多くたまってきたと思います。常に実行ファイルの名前を a.out で作成していると、完成した課題の実行ファイルを残しておくことができず、実行結果を再確認するたびに毎回コンパイルし直すことになります。実行ファイルをプログラムごとに名前を違えて残しておけばそうした面倒がありません。（そうしたことが理由で実行ファイル名をプログラムと同じ（ソースの拡張子を取ったもの）にする慣習ができたのでしょう。）

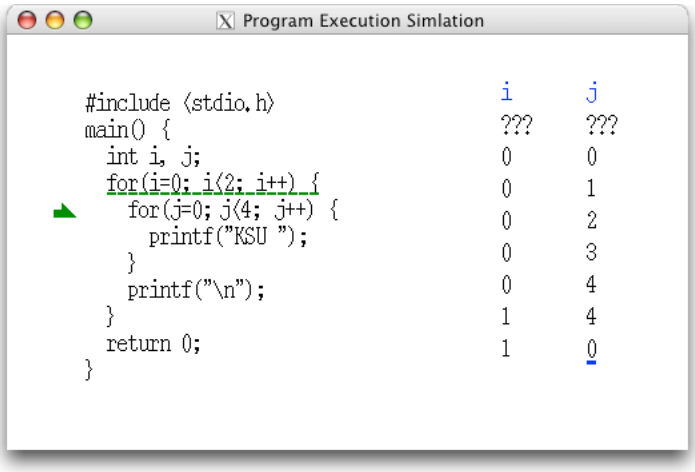
名前を指定しなかった場合と指定した場合で、出来上がる実行ファイルの中身に違いはありません。単に名前が a.out でないだけですので、ファイル名を指定せず a.out の名前で実行ファイルを作成し、後に mv a.out hello などとしてファイルの名前を変更しても同じ事です。
ただ多くのプログラマは –o オプションをつける方法を使っています。

プログラミング演習 A 教材 (#6.5)　ループ復習のためのシミュレータ

プログラムは指示に従って一行一行、順序よく実行されています。これを感じてもらうために、プログラム実行のシミュレータを作ってみました。実行するには、以下のようにしてください。

```
% cd ~yasuda/sim        ← カレントディレクトリをここに移動
% ./trace1              ← trace1 というプログラムを実行
```

起動すると右図のような画面が表示されます。操作方法は以下の通りです。



1. このウィンドウをクリックし、画面の一番上に表示させる。
2. Enter キーを押すごとに、一行ずつ実行される。このときプログラムのどの部分を実行し、変数の値がどのように変化するか図示される。
3. 一行の実行が終わると、次にどこを実行するかが右矢印で示され、そこで停止して次の Enter キーのタイプを待つ。

なお、プログラムが printf( ) を実行すると、 ./trace1 を実行したターミナル（Kterm）の画面上に、"KSU" といった文字が表示されます。この出力とシミュレータウィンドウを見比べながら、プログラムの動きを感じて下さい。

以下のようなプログラムが用意されています。
trace1 - while() を使った簡単なプログラム（1-3 までの数字を書く）
trace2 - for() を使った二重ループ（"KSU" を 2 行 4 列に並べる）
trace3 - while() を使った九九の表（9x9 までやらず、2x3 に縮めています）
trace4 - do while() を使った九九の表（9x9 までやらず、2x3 に縮めています）

こうしたシミュレーションはもちろん紙とペンでも実行できます。プログラムの動きが何かおかしい、納得できないというような状況になったら、ぜひ自分の手元でプログラムの動きと、重要な変数の値を追いかけながらシミュレーションを試みて下さい。

□ 課題 1.

以下のように、上下左右が逆転した九九の表（左上が 81 で右下が 1 となっている）を出力するプログラムを for, while, do while のいずれかを使った二重ループで作って下さい。

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

プログラミング演習 A 教材 (#7)　＝コンマ演算子について＝

■ コンマによる計算式の区切り

右のプログラム例を見て下さい。これは九九の表を出力する課題の回答例なのですが、内側の while ループの後にある記述、

```
i++, j=1;
```

に注目してください。

```
main()
{
  int i=1, j=1;

  while (i < 10) {
    while (j < 10) {
     printf("%3d", i*j);
     j++;
    }
    i++ , j=1;
    printf("¥n");
  }
  return 0;
}
```

注目

これは「i に 1 加算する」処理と「j を 1 に設定する」処理を順次行うことを意識して書かれたものと思います。しかし本来その処理は以下のように書くべきです。

```
i++; j=1;
```

つまり「,」（コンマ）ではなく「;」（セミコロン）で区切られるべきです。

□ セミコロンの意味

C 言語において「;」は、「文の終わり（区切り）」を意味します。逆にプログラム中の改行は C 言語ではほとんどの場合意味を為しませんから、以下の記述はどれも同じ意味です。
（一行の場合も二行の場合もあるが、どれも同じ意味を持つ二文です。）

```
i++; j=1;
```

```
i++;
j=1;
```

```
i++; j=
1;
```

逆にセミコロンなしで「 i++ j=1; 」と書いた場合、これは C の文法としてはおかしい（解釈できない）としてコンパイル時にエラーとなるでしょう。

□ コンマの意味

ところが冒頭の例のプログラム（つまりコンマで接続した記述「 i++, j=1; 」 ）はコンパイル時にエラーとなりません。それどころか正しく動作し、九九の表を出力します。
これは C 言語では「,」には「;」とは異なる意味（別の機能）が割り与えられているためです。

・宣言文である int i,j; や printf(“%d\n”, i); など関数の括弧内に登場する「,」は変数や式の区切りとして扱われます。
（宣言文に int i, j=0; など代入の = が付いていた場合も同じく。）

・それ以外の場所で登場する「,」は単なる区切りではなく、演算子として扱われます。その機能は「,」の左右にある式（計算式、代入式、関数など）を左から順番に実行して、右側の計算結果を最終の結果とする、というものです。

前者はともかく、後者の「コンマ演算子（演算子としてのコンマ）」の意味や動作を把握するのは少し難しいでしょうから、今はとにかく演算子としてのコンマは使わないことを強く勧めます。

興味のあるひとは C 言語の文法書を見て、コンマ演算子の働き、演算子としての優先順位、式文、といったことを調べてみると良いでしょう。「 a=b=c, d=e; 」と「a=(b=c, c=d);」の違いが理解できると思います。（道のりはそう簡単ではありませんが、、、、）

プログラミング演習 A 教材 (#7)

■ グラフィクス、条件分岐

これ以降、EGGX を利用したグラフィクス・プログラミング（画像を描くプログラムを書く）をとりあげます。(C プログラミングガイド EGGX の章を参照)
以下のプログラムを入手して実行してください。

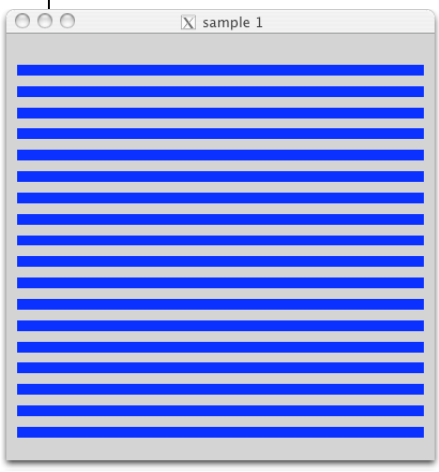
グラフィクスプログラムをコンパイルする場合は、cc ではなく egg コマンドを使います。
（egg コマンドによるコンパイルでも –o オプションによる実行ファイル名の指定が可能です。付けないばあいは a.out の名前で作成されます。）

```
axt22029(84)% egg –o egsample egsample.c
gcc -O2 -Wall -oegsample egsample.c -I/usr/bin -L/usr/bin -I/usr/X11R6/include
 -L/usr/X11R6/lib -leggx -lX11 -lm
axt22029(85)% ls
egsample  egsample.c
axt22029(86)% ./egsample
```

実行すると画面上に絵が表示され、何かキーを押すと終了することがわかるでしょう。

```
/*
 EGGX を使ったサンプル  473088 榎田裕一郎
*/
#include <stdio.h>
#include <eggx.h>

int main() {
  int win;
  float x,y;

  win=gopen(400,400);  /* 描画ウィンドウを開く */
  winname(win, "sample 1"); /* 名前をつける */

  newpen(win, 4);  /* 描画する色を設定 */

  x=10.0; y=20.0; /* x,y の初期座標位置 */

  while( y < 380.0 ) {
    fillrect(win, x, y, 380.0, 10.0);
    y+=20.0;
  }

  ggetch(win); /* キー入力を待つ */
  gclose(win); /* 描画ウィンドウを閉じる */

  return 0;
}
```

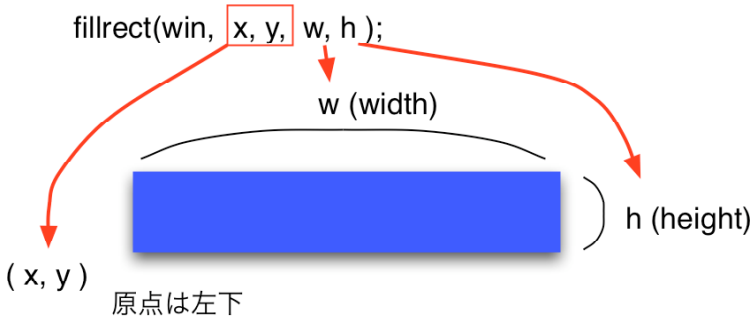
gopen( ) や fillrect( ) などのグラフィクスを描画するために使っているものは、printf( ) などと同じく「関数 (function) 」と呼ばれています。



押さえて欲しいポイント：

・#include <eggx.h> で EGGX に関する準備をする。
・gopen (400, 400) で 400 x 400 画素の描画ウィンドウを開く。
・newpen( ) で色を指定する。色番号は 0 – 15 が設定可能で、それぞれの色は
　0:黒 1:白 2:赤 3:緑 4:青 5:シアン 6: マゼンタ 7:黄 8:灰色(暗) 9:灰色
　10:赤(暗) 11:緑(暗) 12:青(暗) 13:シアン(暗) 14:マゼンタ(暗) 15:黄(暗)
・newpen(win, 4) などのグラフィクス関数の先頭にある引数 win はお決まりなので気にしない。

・fillrect( ) で長方形を描く。引数は左から描く長方形の位置（左下カドの座標）を x, y の順に、次に幅、高さを示す。（右図）
塗りつぶしの色は fillrect( ) 実行前にnwpen( ) で指定した色。

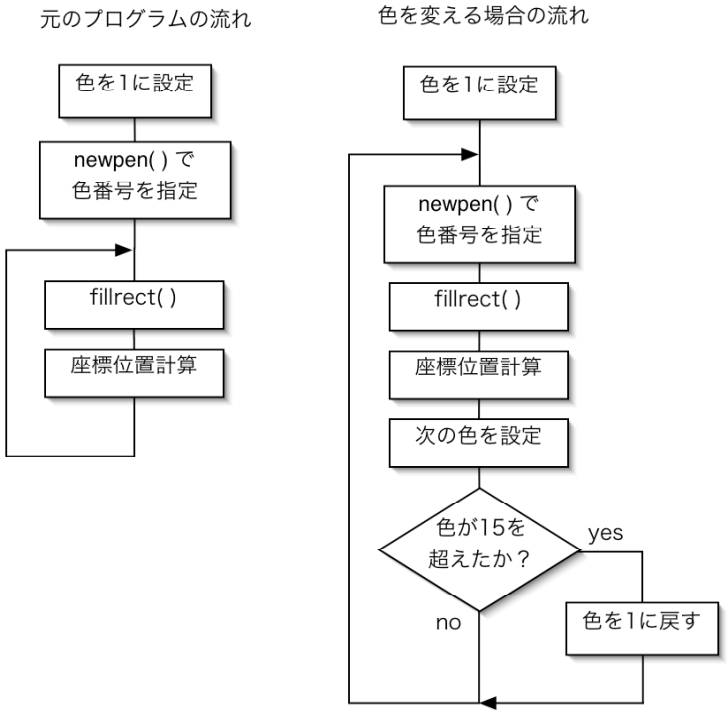


・ggetch( ) でキー入力を待つ。これが無いと「描画ウィンドウを開き、描いて、すぐ消える」ため、結果が見えない。（目に止まらない）
・gclose( ) で描画ウィンドウを閉じる。

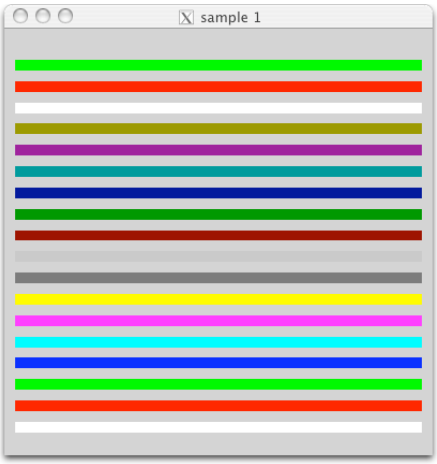
□ 棒の色を変える（条件判断のあるプログラム）

棒の色を一本ずつ変えるようにプログラムを変更します。
考え方：
・newpen( ) で色を変えるのですが、黒（色番号 0 ）は背景と同じ色で見えないので避ける。
・15 色（ 1 – 15 ）使いきったらまた 1 から始めれば良い。

右図を見て下さい。左側がサンプルプログラムそのまま、右側が棒の色を一本ずつ変える場合の処理の流れです。
（while による繰り返し条件判定の部分は省略しています。）



このように何かの条件によって処理の内容を変える場合は、if 文を使います。



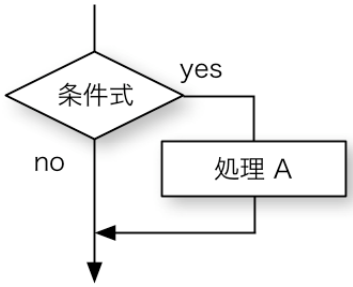
■ 条件分岐

何かの条件によって処理の内容を変える場合は、if 文などを使った条件分岐を設けます。

□ if 文

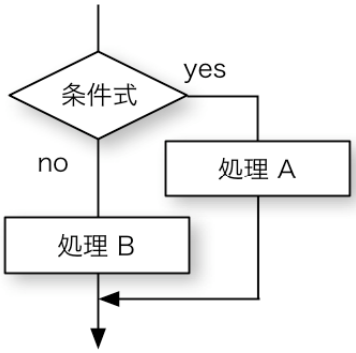
書式：

```
if ( 条件式 ) {
  条件が成立したときの処理 A ;
}
```



または

```
if ( 条件式 ) {
  条件が成立したときの処理 A ;
} else {
  条件が成立しなかったときの処理 B ;
}
```



上記の処理 A, B 部分には条件が成立した（またはしなかった）時の処理を何行でも書けますが、それらが 1 行しか無い場合に限り、{ } を用いず下記のように短く書くこともできます。

```
if ( 条件式 ) 処理 A ;
if ( 条件式 ) 処理 A ; else 処理 B ;
```

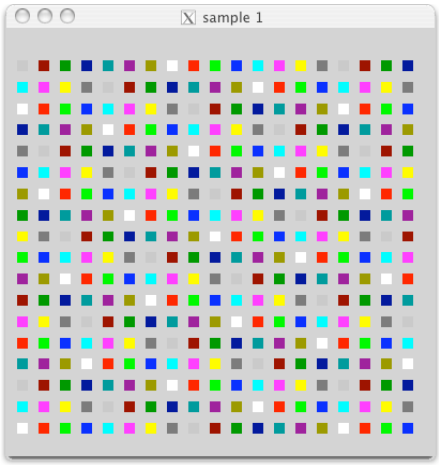
□ 課題 1.

if 文を使って色を変えるようにプログラムを修正して実行してください。
今回は「色番号が 16 になったら 1 に戻す」という条件分岐を設けることになるので、例えば右のような if 文を用いれば良いでしょう。

```
c++;
if( c == 16 ) {
  c=1;
}
```

□ 課題 2.

if 文を使って小さな四角で画面を埋めるようにして下さい。



ヒント：
・まず x 座標を変化させながら小さな四角を描く。
・x が右端に達したら x を 10.0 に戻し、y を変化させる。
・y が上端に達したら終了。

最後の y が上端に到達したら終了、という処理は元のサンプルプログラムに組み込まれているため、今回は考慮しなくても良い筈。

注意：
if 文を使って同じ処理をさせる場合でも、さまざまな書き方がある点に注意。プログラムにはさまざまな書き方があり、唯一の解というものは無い。

■ 条件式　(if 文や whie( ), for( ) などで使えます。)

□ 比較などを行う関係演算子

数値などの大小比較について以下のような演算子が利用できます。

演算子	意味	利用例
==	等しい	`if (a==b) { ... }`
!=	等しくない	`if (a!=b) { ... }`
>	左辺が大きい	`if (a>b) { ... }`
>=	左辺が等しいか大きい	`if (a>=b) { ... }`
<	左辺が小さい	`if (a<b) { ... }`
<=	左辺が等しいか小さい	`if (a<=b) { ... }`

==, != を等値演算子、>, < などを関係演算子と呼んでいます。
これらより算術演算子のほうが優先度が高いため、
if(a < b-1) という記述は
if(a < (b-1)) と同じとみなされます。
（左から順に処理されてまず a < b が先に処理されるようにはなりません。）

□ 複数の条件を並べる論理演算子

複数の条件を並べて判定したい場合は以下のように書きます。

演算子	意味	利用例
&&	AND （両方の条件が成立したら）	`if (a==b && a<100) { ... }` （a と b が等しく、かつ a が 100 未満なら真）
||	OR （どちらかの条件が成立したら）	`if (a==b || a<100) { ... }` （a と b が等いか、または a が 100 未満なら真）
!	NOT （条件の反転）	`if (! a==b) { ... }` （a と b が等しくなければ真）

&& や || を論理演算子と呼んでいます。否定の ! は否定演算子と呼ばれています。
これらは関係演算子よりさらに低い優先度が設定されているので、
if(a < b && c > d) は
if((a < b) && (c > d)) として処理されます。
また、
if(a < b-1 && c + 2 > d -5) は
if((a < (b-1)) && ((c + 2) > (d -5)) として処理されます。

（なるべくバグを発生させない、プログラマの勘違いを誘発させないようにするために、暗黙の優先順位に依存した複雑な論理式を書くより、( ) を明示的に使ってわかりやすい記述を心がける方がよいでしょう。）

■ break , continue, 無限ループ

□ break 文

ループを途中で打ち切りたい、何か条件が成立すればループから脱出したいような場合には break 文が便利です。for, while, do~while によるループの中で break 文を実行すると、その時点でループを打ち切ります。(switch 文でも break を使いますが、これについては次週)

通常、break は if 文と組み合わされて利用されるでしょう。一つ簡単な例を示します。
以下の左側プログラムは「1～10 までの整数の中で 3 の倍数を表示する」ループです。これを break を使って「1～10 までの整数の中で最初に見つけた(最小の) 3 の倍数だけを表示する」ように変更しました(同右側)。実行結果から、break によって最初に倍数を見つけた時点でループから脱出して処理を終了しているのが分かるでしょう。

```
for(i=1; i<=10; i++) {
  if( i%3 == 0 ) {
    printf("I found it! (%d)\n", i);
  }
}
```

実行結果:
```
I found it! (3)
I found it! (6)
I found it! (9)
```

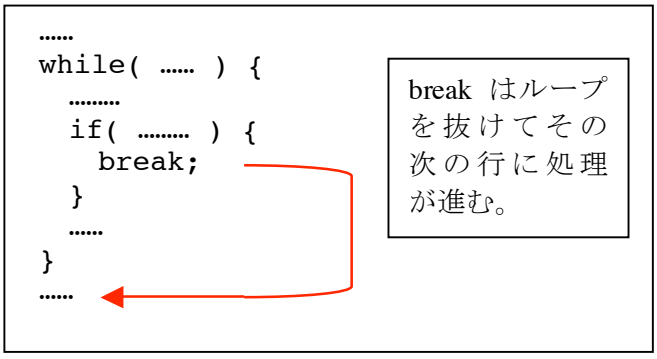
```
for(i=1; i<=10; i++) {
  if( i%3 == 0 ) {
    printf("I found it! (%d)\n", i);
    break;
  }
}
```

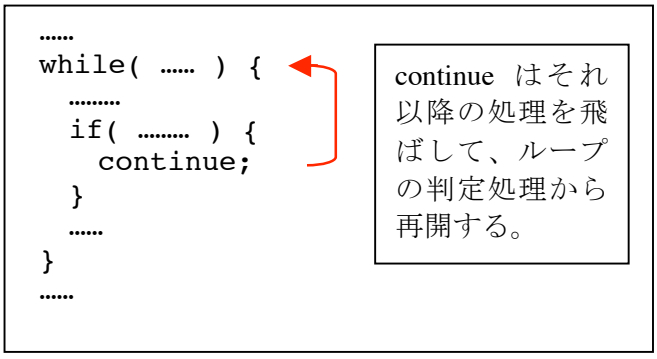
実行結果:
```
I found it! (3)
```

もし break が多重ループの中にあった場合は、その break が含まれる最も内側のループから脱出します。

□ continue 文

continue 文はループ内の処理をそこで止め、次回のループの繰り返し判定処理から続けるものです。(for の場合は判定の前に繰り返し毎処理を行いますので、そこから続けることになります。)

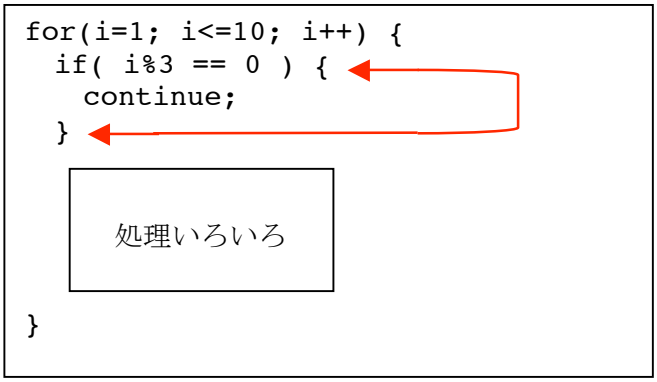


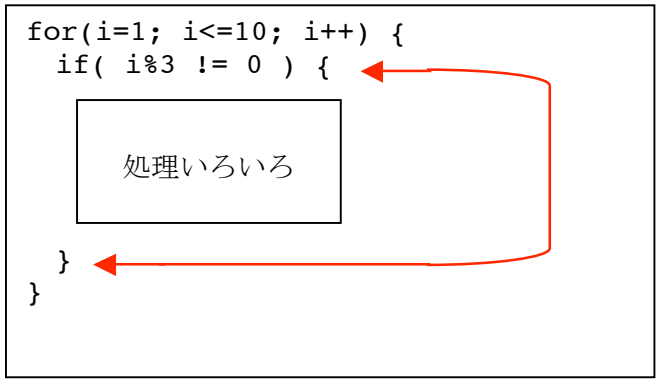


再開は次回ループのための判定処理からです。while や for の場合は上右図のように処理が上に戻るイメージになりますが、do~while の場合は判定がループ末尾にありますから、下まで飛ばすような形になります。

continue を使うケースはそう多くなく、ループの最初に「y が負なら処理不要」といった除外条件をそれと分かりやすく書くために使われる場合などに有効です。

以下に例を示しておきます。左右共に同じ振る舞いをするプログラムですが、continue を使って記述した左の方が、冒頭の if 文のことを考慮しなくてはいけない範囲（矢印でマークした部分）が短くなっています。右側のように記述した場合、if 文の終わりが遠くなり、何十行も間に入ってしまうと分かりにくくなってしまいます。



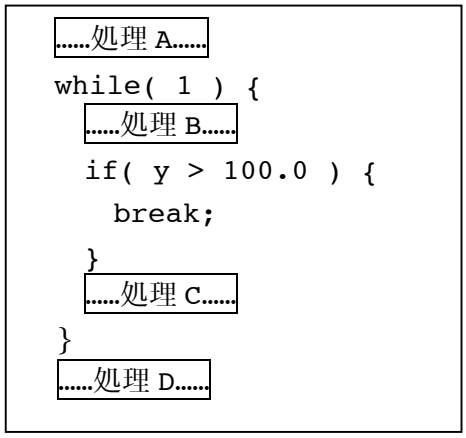


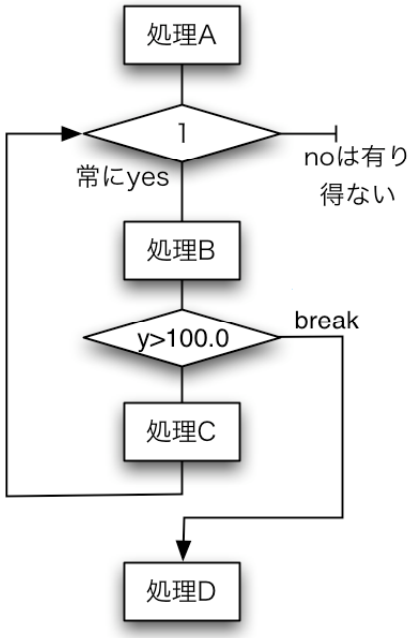
□ 無限ループ

C 言語では条件式の結果として、条件が成立すれば（真の場合は）1 が、不成立の場合（偽の場合）は 0 が得られます。つまり a が 5 の時 if(a<10) {…} は if(1) {…} と同じことになります。
これを利用して while(1) { …… } のように無限に繰り返すループを作る場合もあります。（条件式が 1 つまり常に真なので、この while ループは終わることがない。）

while, for, do ~ while などのループからは break によって脱出することができるため、無限ループはよく if 文及び break 文と組み合せた形で使われます。

以下に while(1) を使った無限ループのプログラム例を示します。もし y が 100.0 を越えたら、という条件で break する部分に注目して下さい。



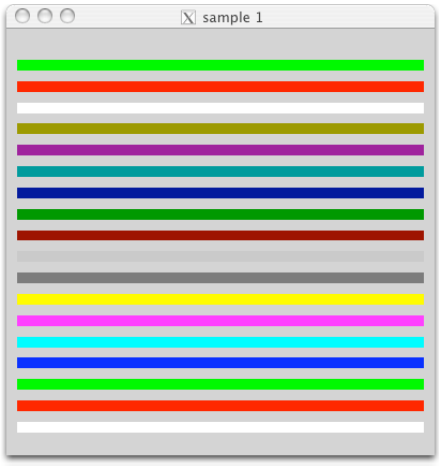


プログラミング演習 A 教材 (#7) プログラム例

□ 課題 1.

if 文を使って色を変える。

```
c=1;
while( y < 380.0 ) {
  newpen(win, c);
  fillrect(win, x, y, 380.0, 10.0);
  y+=20.0;
  c++;
  if( c == 16 ) {
    c=1;
  }
}
```



まず色番号の変数として int 型変数 c を設けます。
最初に c=1 で色番号を 1 としておき、ループごとに c++; で次の色番号を計算します。色番号が 15 になるまでは if 文は成立しませんが、色番号 15 で四角を描いたあとは c++ によって色番号が 16 となり、if 文が成立して色番号 c には 1 が設定され直して、ループを継続します。

たとえば if 文の else 節を使って以下のように書くことも出来ます。

```
c=1;
while( y < 380.0 ) {
  newpen(win, c);
  fillrect(win, x, y, 380.0, 10.0);
  y+=20.0;
  if( c == 15 ) {
    c=1;
  } else {
    c++;
  }
}
```

この例では、(前のプログラム例のように) 次の色を計算して c が 16 になったことをチェックするのではなく、

・今使った色が 15 だったかどうかをチェックし、
・15 だったら 1 に戻し、
・15 でなければ c++ で次の色を計算しています。

```
if( c != 15 ) {
  c++;
} else {
  c=1;
}
```

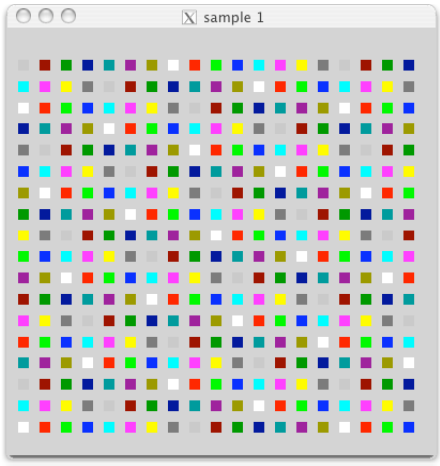
左のように条件判定を逆論理にして、if のブロック内の記述を else 節と入れ替えて書いても良いでしょう。

注意：
if 文を使って同じ処理をさせる場合でも、さまざまな書き方がある点に注意。プログラムにはさまざまな書き方があり、唯一の解というものは無い。より分かりやすい記述を心がけることが重要。
(そもそもこの例題ならば if 文を使わない書き方も幾通りも有り得る。)

□ 課題 2.

if 文を使って小さな四角で画面を埋める

```
x=10.0; y=20.0;
c=1;
while( y < 380.0 ) {
  newpen(win, c);
  fillrect(win, x, y, 10.0, 10.0);
  x+=20.0;
  if( x > 380.0 ) {
    x=10.0;
    y+=20.0;
  };
  c++;
  if( c == 16 ) {
    c=1;
  }
}
```




```
  if( x > 360.0 ) {
    x=10.0;
    y+=20.0;
  } else {
    x+=20.0;
  }
```

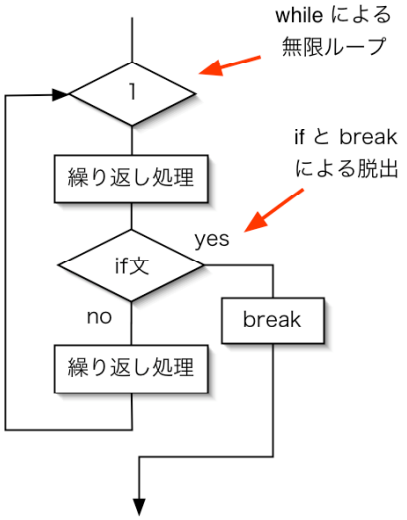
この例も左のように else 節を使って課題 1. のように「計算する前にチェックし、端にきていたら x 座標を左に戻し、y 座標を加算する」というようにもできます。
（ x の条件判定を 380.0 から 360.0 に変更して x が右に行きすぎないようにしている点にも注意。）

□ 無限ループを用いた例（参考）

C 言語では条件式の結果として、条件が成立すれば（真の場合は）1 が、不成立の場合（偽の場合）は 0 が得られます。
これを利用して while(1) { …… } のように無限に繰り返すループを作る場合もあります。（条件式が 1 つまり常に真なので、この while ループは終わることがない。）

```
x=10.0; y=20.0;
c=1;
while( 1 ) {
  newpen(win, c);
  fillrect(win, x, y, 10.0, 10.0);
  x+=20.0;
  if( x > 380.0 ) {
    x=10.0;
    y+=20.0;
    if( y >= 380.0 ) {
      break; /* ループから脱出 */
    }
  }
  c++;
  if( c == 16 ) {
    c=1;
  }
}
```

break 文によって while, for, do ~ while などのループから脱出することができるので、これを if 文と組み合わせています。



プログラミング演習 A 教材 (#8)

■ 入力、scanf、if、switch.

pinrtf ( ) などによるプログラムからの結果の取り出し（人間に見える形にする）を出力と呼んでいます。これに対してプログラムに値や指示を与えることを入力と呼びます。
ここではそうした方法のうちの一つ、キーボードから文字入力を行う例を示します。

□ scanf による文字入力

右のプログラムを入力して実行してください。（グラフィクスプログラムではないので、コンパイルは egg コマンドではなく cc コマンドで良い。）

```
#include <stdio.h>

/*
  消費税計算をする 473088 榎田裕一郎
*/

main() {
  int price, total;

  scanf("%d", &price);
  total=price * 1.05;
  printf("total = %d\n", total);

  return 0;
}
```

実行するとキーボードからの入力を待つ状態になります。以下のように 230 と入力すると、消費税を計算して結果を画面に出力します。

```
% ./a.out
230
total = 241
%
```

`scanf("%d", &price);` で price 変数に入力された値（この例では 230 ）が入ります。

書式：
`scanf(”書式”, &変数 1, &変数 2, … &変数 n);`

”書式”には入力する変数の数だけ % に続けて入力する形式を %d や %f の変換文字列で指定します。変換文字列は printf のそれと同じで、%d は十進数の整数、%f は float 型の実数に、%lf が double 型の実数に対応します。変数の前の & については今は説明しません(*1)。

つまり二つの変数に値を設定したい場合は、以下のように書きます。
```
scanf("%d %f", &price, &tax);
```

二行に分けて書く方法もあります。
```
scanf("%d", &price);
scanf("%f", &tax);
```

どちらの方法でも、実行時の入力の方法は同じです。二行に分けて入力することもできます。

```
% ./a.out
230 1.05
total = 241
%
```

```
% ./a.out
230
1.05
total = 241
%
```

□ 課題 1.

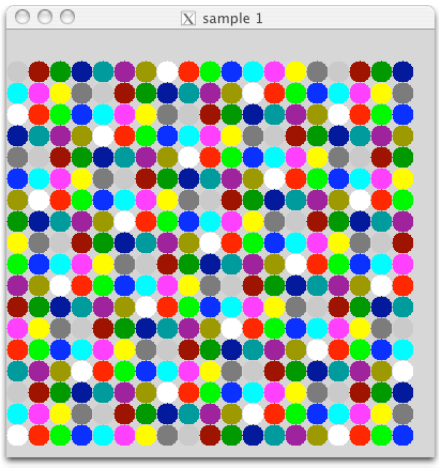
上記の例のように税率も scanf で入力するようにして実行し、動作を確認してください。

*1 &記号、つまりポインタについては「プログラミング B」で説明します。

□ 課題 2.

入力に応じて処理を変える

先週の小さな四角を描くプログラムに手を加えて、入力に対応して四角ではなく丸を描くように機能追加してください。



実行すると以下のように 1 か 2 のどちらかを入力するように求め、1 を入力すれば四角、2 を入力すれば円を描くようにしてください。(右図は 2 を指定した場合)

```
% ./a.out
drawing type? (1-2) = 2
%
```

以下の点に注目してプログラムを修正して下さい。

押さえて欲しいところ:

・printf( ) でプロンプトとなる文字列を出力。(改行なし)
・scanf( ) で描画図形のタイプを入力し、int 型変数 type に格納。
・type の値によって描く図形のタイプを if 文で変更。
・fillrect( ) で四角形(塗りつぶし)を描く。
・fillarc( ) で円(塗りつぶし)を描く。

修正の例

```
printf("drawing type? (1-2) = ");
scanf("%d",&type);
.. (中略) ..
  if(type == 1) {
    fillrect(win, x, y, 10.0, 10.0); /* 四角を描く */
  } else {
    fillarc(win, x, y, 10.0, 10.0, 0.0, 360.0, 1); /* 円を描く */
  }
```

fillarc( ) 関数の引数の役割は以下の通りです。(詳細な仕様は <<ガイド 9.3>> 参照)

```
fillarc(win, x, y, w, h, s, e, d);
```

x, y : 描く円の中心の座標位置 (x , y)
w : 描く円の水平方向の径
h : 描く円の垂直方向の径
s, e : 描きはじめの角度と、描き終わりの角度。0.0 と 360.0 で円に、それ以外で扇形になる。
d : 描く方向(右回りが 1 で左回りが –1)

d は整数型、それ以外は全て実数、float 型です。

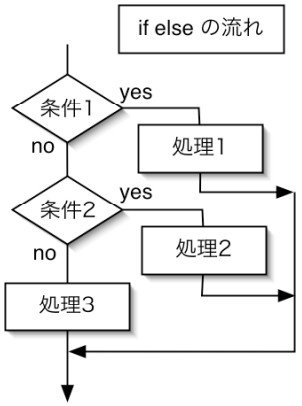
□ さまざまな if 文の書き方

1, 2 以外の値を入力した場合の例外処置を下記のように含めました。1 でも 2 でもない場合は描画ウィンドウを閉じてプログラムを途中で終了します。else if 句の使い方に注目して下さい。

```
    if(type == 1) {
      fillrect(win, x, y, 10.0, 10.0); /* 四角を描く */
    } else if(type == 2) {
      fillarc(win, x, y, 10.0, 10.0, 0.0, 360.0, 1); /* 円を描く */
    } else {
      printf(" 1 か 2 で指定してください。\n");
      gclose(win);
      return 0;
    }
```

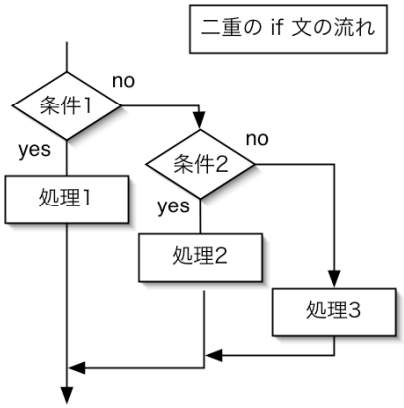
（下の例では一つですが、else if は幾つでも続けて書けます。）

```
if(条件 1) {
  処理 1;
} else if(条件 2) {
  処理 2;
} else {
  処理 3;
}
```



同じ意味の処理を if 文を入れ子にして表現することもできます。入れ子も何重にでもできます。

```
if(条件 1) {
  処理 1;
} else {
  if(条件 2) {
    処理 2;
  } else {
    処理 3;
  }
}
```

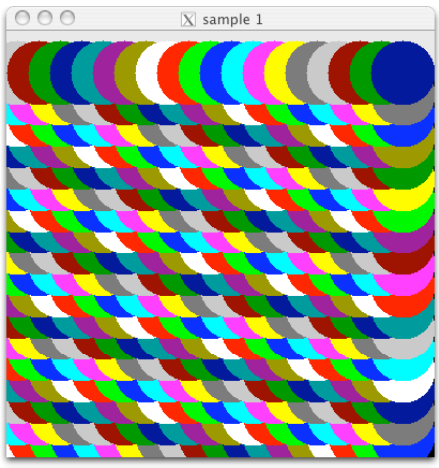


□ 課題 3.

・上記の例外処置を組み込み、
・type だけでなく、長さも入力し、
・四角形の辺の長さ、円の径をその長さで描画する。

つまり以下のような実行時の入力によって右図のような結果となるように。（プロンプトに続いて 2　30.0 と入力した。）

```
% ./a.out
drawing type (1-2) and size = 2 30.0
%
```



□ switch 文による場合分け

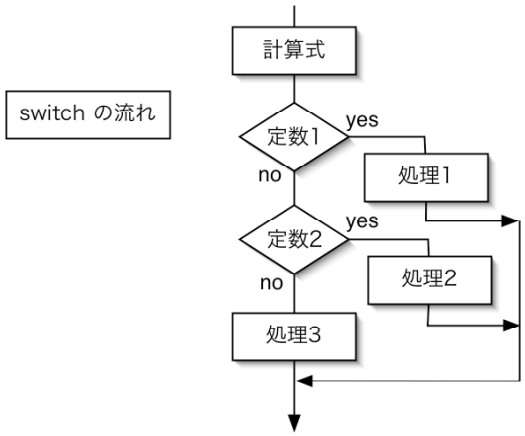
if else を使った条件分岐を、switch 文を使ってより分かりやすく記述できる場合があります。

```
switch( type ) {
case 1:
  fillrect(win, x, y, w, w); /* 四角を描く */
  break;
case 2:
  fillarc(win, x, y, w, w, 0.0, 360.0, 1); /* 円を描く */
  break;
default:
  printf(" 1 か 2 で指定してください。\n");
  gclose(win);
  return 0;
}
```

switch( ) のカッコ内に変数や計算式を書き、その結果がどのケースに該当するかによって処理を分岐させます。

書式：

```
switch( 式 ) {
case 定数 1:
  処理 1;
  break;
case 定数 2:
  処理 2;
  break;
..（略）..
default:
  処理 n;
}
```



注意点：

・case に続く条件部分には定数または定数式しか書けません。（ x < 10 など条件式は不可。）
・case による分岐は幾つでも書けます。
・どのケースにも該当しなかった場合は default: に続く処理を実行します。
・default は省略できます。（その場合該当しないケースでは何もせずに swtich を通過します。）
・各処理の最後には break; を付けてください(*1)。break はループを脱出するものとして
　登場していましたが、switch からの脱出にも使います。
・switch の直後の { と、対応する } を忘れないように。

□ 課題 4.

サンプルプログラムを加工し、switch を使って描く図形を 4 種類指定できるようにしてください。あと二つの図形には（中を塗りつぶさない）四角を描く `drawrect( )` 、（中を塗りつぶさない）円を描く `circle( )` を利用すると良いでしょう。各関数の仕様については<<ガイド 9.3>> を参照。

*1 たとえば一番目の処理の最後に break; が存在しなかった場合、処理 1 が終わると無条件に処理 2 を実行します。なぜこのような言語仕様になっているのかはここでは説明しません。興味のある人はコンパイルされたコードを読むと分かるでしょう。

プログラミング演習 A 教材 (#9) ＝これまでに出てきた機能などのまとめ＝

□ データの型

char : 文字(character)を扱うための型。'x', 'a' など。
int : 整数(integer)を扱うための型。123, 0, -3 など。
float , double: 実数を扱うための型。123.456, 0.0001, 0.0, 1024.0 など。

123.456 は実数でも double 型。float 型と明記したい場合は 123.456f と最後に f をつける。

□ 代表的な printf の変換文字列

	意味	使用例	その結果
%c	文字を表示	printf("[%c]¥n",'a');	[a] 文字がそのまま表示される。
%d	整数を表示	printf("[%d]¥n",10);	[10] 桁数不定
		printf("[%5d]¥n",10);	[10] 5 桁で表示。不足分は空白。
		printf("[%05d]¥n",10);	[00010] 5 桁。不足はゼロで埋める。
%f	実数を表示	printf("[%f]¥n",12.345);	[12.345000] 桁数不定
		printf("[%9.5f]¥n",12.345);	[12.34500] 小数点含めて全体が 9 桁、小数以下が 5 桁。
%e	実数を表示	printf("[%e]¥n",12.345);	[1.234500e+01] 浮動小数点で表示

□ ループ for 文、while 文、do ~ while 文

```
for( 開始時処理 ; 繰り返し条件式 ; 繰り返し毎処理 ) {
    繰り返す処理
}
```

```
while( 繰り返し条件式 ) {
    繰り返す処理
}
```

```
do {
    繰り返す処理
} while( 繰り返し条件式 );
```

□ scanf 文

```
scanf("書式", &変数 1, &変数 2, … &変数 n);
```

"書式"には入力する変数の数だけ % に続けて入力する形式を変換文字列で指定。
%d が十進数の整数、%f は float 型の実数、%lf は double 型の実数に対応。

□ 条件式

演算子	意味	利用例
==	等しい	if (a==b) { ... }
!=	等しくない	if (a!=b) { ... }
>	左辺が大きい	if (a>b) { ... }
>=	左辺が等しいか大きい	if (a>=b) { ... }
<	左辺が小さい	if (a<b) { ... }
<=	左辺が等しいか小さい	if (a<=b) { ... }

演算子	意味	利用例
&&	AND （両方の条件が成立したら）	if (a==b && a<100) { ... } （a と b が等しく、かつ a が 100 未満なら真）
||	OR （どちらかの条件が成立したら）	if (a==b || a<100) { ... } （a と b が等いか、または a が 100 未満なら真）
!	NOT （条件の反転）	if (! a==b) { ... } （a と b が等しくなければ真）

□ if 文

```
if ( 条件式 ) {
  条件が成立したときの処理 ;
}
```

```
if ( 条件式 ) {
  条件が成立したときの処理 ;
} else {
  条件が成立しなかったときの処理 ;
}
```

```
if(条件 1) {
  条件 1 が成立したときの処理;
} else if(条件 2) {
  条件 2 が成立したときの処理;

  中略： else if 文は幾つでも書ける

} else if(条件 n) {
  条件 n が成立したときの処理;
} else {
  条件 1 … n が成立しなかったときの処理;
}
```

最後の else 節は省略可

□ switch 文

```
switch( 式 ) {
case 定数 1:
  処理 1;
  break;
case 定数 2:
  処理 2;
  break;
..（略）..
default:
  処理 n;
}
```

・case に続く条件部分には定数または定数式のみ。

・default は省略可能。（該当しないケースでは何もせずに swtich を通過。）

・各処理の最後には break; が必要。

プログラミング演習 A 教材 (#9)

■ 関数

プログラムのなかの一部分を機能ごとに切り出し、関数として分けて記述することができます。これによってより分かりやすいプログラムの記述ができます。またプログラムを機能ごとの単位に仕分けて書くことによって、機能単位の再利用が容易になり、共通の関数を使った別のプログラムを簡単に作れるようになります。大規模なプログラムを作る重要な手法の一つです。

□ 簡単な関数の例

以下に階乗 (factorial) を計算するプログラムの例を示します。入力し、実行して、たとえば 5 の階乗が正しく 120 (5 * 4 * 3 * 2 * 1) になることを確認して下さい。

入力処理

計算処理

出力処理

```
#include <stdio.h>
int main() {
  int n,i,fac;

  printf("n=");
  scanf("%d", &n);

  fac=n;
  for(i=n-1; i>1; i--) {
    fac*=i;
  }

  printf("factorial = %d\n", fac);

  return 0;
}
```

11 以上の階乗を求めようとすると桁あふれを起こして正しい結果が出ないので注意。

プログラムをよく見ると、入力処理、計算処理、出力処理に分けられることがわかります。
今回はこのうち計算処理の部分だけを別の関数にします。

階乗の計算処理だけを factorial という名前の関数に分けて書いた例を以下に示します。二つのプログラムの関係をメインルーチン、サブルーチンと表現することもあります。

```
int main() {
  int n, fac;

  printf("n=");
  scanf("%d", &n);

  fac=factorial(n);

  printf("factorial = %d\n", fac);

  return 0;
}
```

呼び出し

```
int factorial(int n)
{
  int i,f;

  f=n;
  for(i=n-1; i>1; i--) {
    f*=i;
  }

  return f;
}
```

戻り

メインルーチン（main 関数）　　サブルーチン（factorial 関数）

処理の流れ：
・ プログラムは main( ) の上から順に実行される。
・ main の途中で factorial( ) 関数を呼び出すと、
・ factorial 関数に処理が移る。
  ・ 階乗の計算を行い、
  ・ return でメインルーチンに処理が戻る
  ・ そのとき return f としたため、階乗の結果が factorial( ) 呼び出しの結果として戻される。
・ メインルーチン側で factorial( ) 関数の結果（戻り値）を fac 変数に代入。
・ そのまま main の処理を続行する。

□ 関数の定義、値の引き渡し方

例では factorial 関数は以下のようにして定義しました。

```
int factorial(int n)
{
  ……
}
```

`int factorial(…)` の先頭の型 `(int)` は、`factorial` 関数の戻り値（後述）の型を示す。

この関数をメインルーチン側から呼び出すときは、例えば以下のようにします。

```
factorial( 値 );
```

値の部分には数値(100 等)や変数(n 等)、それらを組み合わせた計算式などが入ります。この値がサブルーチン側（factorial 関数側）で用意した変数 n に引き渡され、サブルーチン内での実行が始まります。

書法：(メインルーチン側)
関数名`( 値 1, 値 2, … 値 z )`

書法：(サブルーチン側)
型 関数名`( 型 変数 1, 型 変数 2, … 型 変数 z )`

こうした受け渡しに用いるカッコ内の値（や変数）を引数（ひきすう）と呼びます。引数は複数用意できますが、そのときは呼び出す側と呼ばれる側の用意した値と変数が左から順に組み合わされて代入されていきます。引き渡す引数の数と型がきちんと一致していることが重要です。
上の書法では、メインルーチンで引数に設定した「値 2」は、サブルーチンで用意した「変数 2」に設定されますが「値 2」のデータ型は「変数 2」の型に一致していなければなりません。

□ 戻り値

`return` 文の実行によって、処理はサブルーチンからメインルーチンに戻ります。このとき、「 `return 値;` 」と書くことで関数の結果そのものを設定することができます。

つまり呼び出すときに `fac=factorial(n);` のようにしておくと、サブルーチン側の return によって戻された結果（ n の階乗）が変数 fac に代入されます。

注意：
return で戻す値は、関数自体の宣言文 (int factorial(…) の部分)の先頭にある関数自体の型宣言と一致していること。

□ 実際のプログラムの記述

実際の C プログラムのなかでは、サブルーチンはメインルーチンと同じファイルのなかにずらっと並べて書きます。

サブルーチン（呼び出される側）はメインルーチンの呼び出しより前に記述されていなければなりません。

つまりサブルーチンは main より前（上）に書くことになります。

関数から関数を呼び出すこともできますが、そうした場合も常に呼ばれる側がより前（上）に記述されている必要があります。

（例ではメイン、サブで同じ名前の変数を用いていますが、これらは別の名前を用いても構いません。）

```
#include <stdio.h>

int factorial(int n)
{
  int i,f;
  f=n;
  for(i=n-1; i>1; i--) {
    f*=i;
  }
  return f;
}

int main() {
  int n, fac;
  printf("n=");
  scanf("%d", &n);

  fac=factorial(n);

  printf("factorial = %d\n", fac);
  return 0;
}
```

□ main の意味

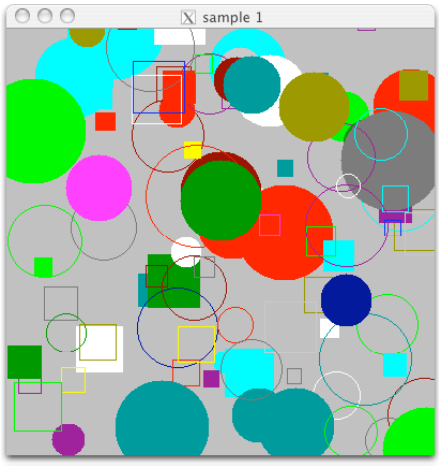
よく見ると main( ) もひとつの関数として機能していることが分かると思います。
実際、C 言語ではメインルーチンは特別な存在ではなく、単に実行時に最初に呼び出される関数がたまたま main( ) という名前に固定されている、というだけのことです。

□ 課題 1.

以前に作った「4 種類の図形を描き分ける」プログラムを、サブルーチンを使ってより読みやすくしてください。（次ページのひな形を見て空白を埋めていけばいいでしょう。）

ヒント：
・draw( ) という関数を作ってみました。
・draw( ) の引数は左から色、図形タイプ、座標位置(x, y)、図形の幅です。
・もし 1-4 以外の図形タイプを設定した場合は、draw( ) の戻り値を 0 以外にします
・dray( ) の戻り値次第で、必要ならエラー処理をしてメインルーチンの処理を中断します。

□ 課題 2.



draw( ) 関数を利用して違う動きをするプログラムを作りましょう。乱数を発生させる rand( ) 関数を利用して、ランダムな位置に 4 種類の図形を 100 個描くプログラムを作って下さい。

乱数発生のためのサンプルプログラムは次ページにあります。

サブルーチンをブラックボックスとして再利用できることを実感できると思います。

課題 1. ひながた

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

int draw(int win, int c, int type, float x, float y, float w)
{
  int rc=0; /* 戻り値をゼロに */

  … 略 …

  return rc;
}

int main() {

  … 略 …

  while( y < 380.0 ) {
    rc=draw(win, c, type, x, y, w); /* 描画 */
    if( rc != 0 ) { /* 戻り値がゼロでなければエラー処理 */
      printf(" 1 から 4 で指定してください。\n");
      gclose(win);
      return 0;
    }

      … 略 …

  }

  ggetch(win); /* キー入力を待つ */
  gclose(win); /* 描画ウィンドウを閉じる */

  return 0;
}
```

課題 2. のための乱数発生サンプルプログラム。(0-99 までの乱数を表示する。)

ポイント:

- 疑似乱数関数 rand( ) のために stdlib.h の include が必要。
- 最初に srand( ) 関数を一度だけ呼び出す必要あり。srand に与える引数はランダムな数値。Seed と言う。
- rand( ) 関数は 0 から 2147483647 までの整数を返す。

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

int main() {
  int seed, num, cnt;

  printf("random seed = ");
  scanf("%d",&seed);
  srand(seed);

  for(cnt=0; cnt<10; cnt++ ) {  /* 10 回ループ */
    num=rand() % 100; /* 余りは 0-99 */
    printf("%d\n",num);
  }

  return 0;
}
```

プログラミング演習 A 教材 (#10)

■ アニメーション

□ 簡単な動画の例

右に簡単な動く絵を描くプログラム例を示します。(#include などは省略しています)

ポイント：
- gclr( ) 関数で画面をクリアする
- その後に描画を行い
- msleep( ) 関数によって少し待ってから画面消去をする

msleep( ) によって待ち時間が設定されていない場合、どのような結果になるか試して下さい。

```
int main() {
  int win;
  float x;

  win=gopen(400,400);
  winname(win, "Animated");

  for(x=10.0; x<380.0; x+=10.0) {
    gclr(win);       /* 画面を消去 */
    fillrect(win, x, 30.0, 10.0, 10.0);
    msleep(50);      /* 少し待つ */
  }

  ggetch(win);
  gclose(win);

  return 0;

}
```

□ 課題 1.

サンプルプログラム bounce.c を取得して実行し、赤い円が壁に向かって進むのを確認してください。次にこのプログラムを改造して、左右の壁に当たったら跳ね返り、上端まで届いたら終了するようにしてください。

□ 課題 2.

それができれば、次は上下の壁も同様に跳ね返るように改造してください。

□ 課題 3.

それができれば、次は何かもう少し異なるアクションを追加してください。
例えば壁に当たるたびに、
・図形の形が変わる
・色が変わる
・大きさが変わる
・スピードが変わる
・進行方向（角度）が変わる
などはどうですか？

おもしろいアニメーションができることを望みます。

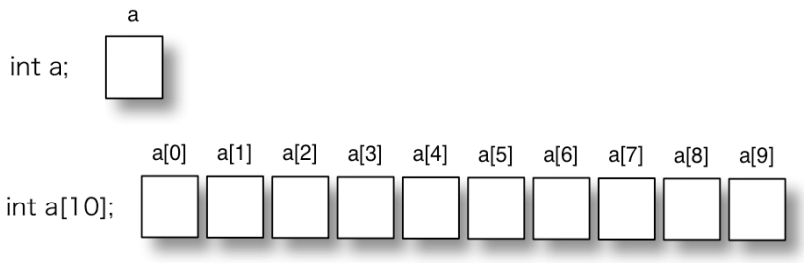
プログラミング演習 A 教材 (#11)

■ 配列

□ 配列変数の宣言

関連性のある同じ型の複数の値を扱うために「配列変数」と呼ばれる仕組みがあります。プログラム中では変数名の後ろに [ ] をつけることで通常の変数と区別して表記します。

int 型の変数として a を宣言するには右図のように int a; と宣言します。これによって一つだけ、整数を代入できる領域が確保されます。
これに対して、int a[10]; と宣言することで、int 型の数値を 10 個扱うための配列変数が宣言できます。



このとき 10 個の整数を代入できる領域が確保され、それぞれの領域は [ ] をつけて番号で指定します。上の例では a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9] と 10 個用意されました。この一つ一つの領域のことを要素と呼び、要素の番号のことを添え字（インデックス）と呼びます。インデックスは 0 から始まっているため、10 個宣言した配列変数の最後の要素が a[10] ではなく a[9] であることに注意してください。

□ 簡単な利用例

乱数を使って 10 個の要素にそれぞれランダムな数値を代入し、後で表示するプログラムを右に示します。

```
int main() {
  int i, a[10], seed;

  printf("random seed = ");
  scanf("%d",&seed);
  srand(seed);

  /* 各要素にランダムな数を代入 */
  for(i=0; i<10; i++) {
    a[i]=rand();
  }

  /* 各要素をプリント */
  for(i=0; i<10; i++) {
    printf("a[%d] = %d\n", i, a[i]);
  }

  return 0;
}
```

ポイント：
- 疑似乱数関数 rand( ) のために stdlib.h の include が必要。
- 最初に srand( ) 関数を一度だけ呼び出す必要あり。srand に与える引数はランダムな数値。Seed と言う。
- rand( ) 関数は 0 から 2147483647 までの整数を返す。

i を 0 から 9 まで変化させながら 10 回ループさせ、a[i] = rand( ); とすることで配列変数の各要素に数値が代入されています。実行して 10 行のランダムな数値が表示されることを確認してください。

□ 課題 1.

下記のプログラム snow.c を取得して実行し、その結果を確認してください。特にアニメーションをどのようにして実現しているかに注目して下さい。

ポイント：
- while() によってループするたびに、
- gclr( ) によって画面を消去し、
- 少し下の場所に雪粒を描き直し、
- msleep( ) によって少しだけ待つ。（引数に指定する待ち時間の単位はミリ秒）

```
int main() {
  int win, seed;
  float x, y, w;
  …… (中略) ……

  while(1) {
    gclr(win);                    /* 画面を消去 */
    fillrect(win, x, y, w, w);  /* 四角を描く */
    y-= w / 2.0;                  /* 雪粒の大きさだけ下に移動 */
    if( y < 0.0 ) {
      x=(float)(rand() % 400); /* x 座標位置を再設定 */
      y=400.0;                   /* y 座標位置は再び上端へ */
    }
    msleep(10);                   /* 少し待つ */
  }

  /* このプログラムは終了しないので return 0; などは無し */
}
```

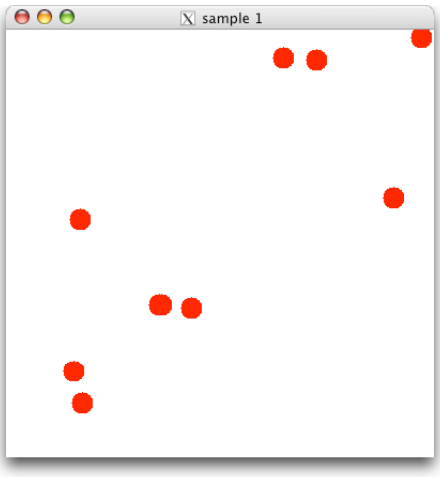
このプログラムを配列変数を使うように修正し、雪粒を 10 個に増やして下さい。

ヒント：
- x, y をそれぞれ配列変数として各雪粒の位置を 10 セット記憶できるようにします。
- 初期位置や下端に到達した判定、fillrect( ) による描画など、雪粒に関するすべての処理をループを使って各要素ごとに行います。

□ 課題 2.

跳ね返る球のアニメーションを描くプログラム pingpong.c を取得して実行し、動作を確認してください。そこでは一つだけ球が動いていると思います。これを右図のように複数の球が跳ね返るように修正してください。

余力のある受講生は球の色を変えるなり、もっと多くの球を出すなり、工夫してより面白いアニメーションを作ってください。



プログラミング演習 A 教材 (#12)

■ 数学関数

□ 数学関数の使用例

C 言語には sin( ) 関数や cos( ) 関数など、数学的な計算を行ってくれる関数がひと揃い用意されています。こういった関数を数学関数と呼んでいます。代表的な関数としては以下のようなものがあります。

sin　正接を求める	sqrt　平方根を求める
cos　余弦を求める	pow　累乗を求める
tan　正接を求める	log　対数を求める
fabs　絶対値を求める	exp　指数を求める

他の関数、詳しい使い方などについては <<ガイド 8.2>> や参考書を参照すると良いでしょう。
以下に sin 関数を例にとって、その使用法を簡単に説明します。

□ 数学関数を使うために

これからプログラムの中で数学関数を使うために、まず決まり文句として先頭に以下の一行を買いて下さい。

#include <math.h>

これは数学関数などを宣言している math.h ファイルを取り込む、という指示なのですが、今は気にせずただ丸覚えして下さい。EGGX によるグラフィクス機能を利用するために冒頭に eggx.h を取り込むように書いたのと同じ事です。
またコンパイルするときに、以下のように -lm オプションが必要になります。

```
% cc sample.c -lm
```

これは数学関数のためのライブラリを使う指示で、これがないと下記のように「sin という名前は未定義 (undefined) である」といったエラーが出るでしょう。

```
% cc –o sample sample.c
/tmp/ccHZhcFU.o: In function `main':
/tmp/ccHZhcFU.o(.text+0x3f): undefined reference to `sin'
collect2: ld returned 1 exit status
%
```

なお egg コマンドでコンパイルするときは –lm は追加しなくてかまいません。理由は egg コマンドが自動的に –lm もつけてくれているからです。
（ egg コマンド実行時のメッセージをよく見ると –lm と付いているのがわかるでしょう。）

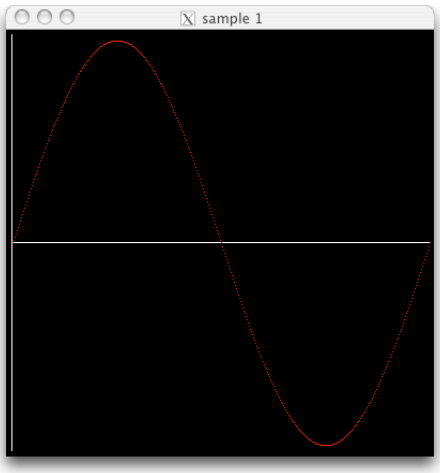
これで y = sin(r); のようにして使えば y に角度 r の時の正接を計算してくれます。

ただし sin 等の三角関数に与える引数は、
・ double 型の実数であること
・ 角度は、ラジアン単位（弧度）で指定することに注意して下さい。
sin 関数の戻り値は double 型で返されます。

ラジアン単位（弧度法）では度数法で言うところの 360 度を 2 πラジアンとしています。
つまり sin 関数に 60 度に相当する角を渡したければ、
60 / 360 * 2 * 3.141592 ≒ 0.018278
を与えることになります。

□ 課題 1.

右図のような sin カーブを描くグラフを描いてください。



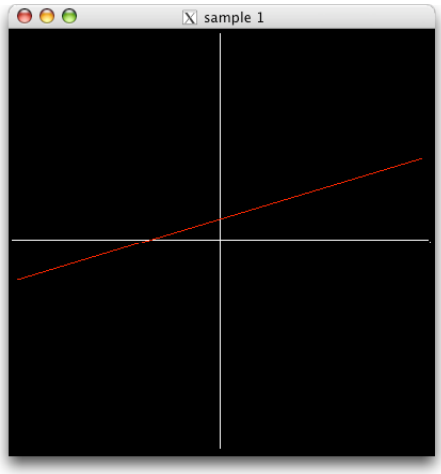
参考として画面上に点を描く pset( ) 関数を使って一次関数を描くサンプルプログラムを以下に付けておきます。

□ サンプル（一次関数のグラフを描く）

以下に $y = ax + b$ $(a=0.3, b=20.0)$ の一次関数グラフを描くプログラムを示します。

押さえて欲しいポイント：
- line( ) 関数で座標軸を描いています。第四引数に PENUP と書くことで始点まで移動し、そこから PENDOWN と書いて与えた座標位置までを線引きします。
- pset( ) 関数で、計算した座標位置に点を打っています。
- 原点をグラフィクスウィンドウの真ん中（200, 200）に置いているので、pset( ) 関数の x, y 座標位置指定にはそれぞれ 200.0 を加算しています。

プログラムと実行結果

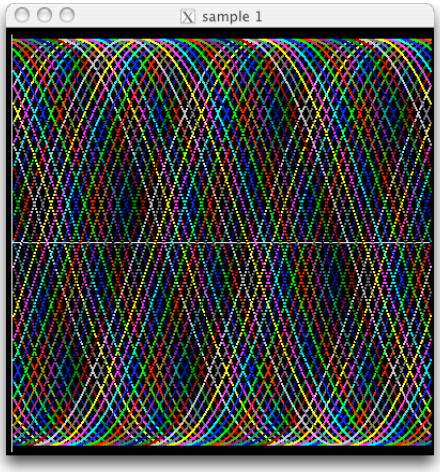


注意：
pset( ) など EGGX の描画関数群の引数が float 型であるため、y 自身の型や y=ax+b を計算する式は float 型で書いています。sin( ) 関数の引数は double 型ですので、y=sin(r) (r は角度)を求める時の r は double 型で与えて下さい。

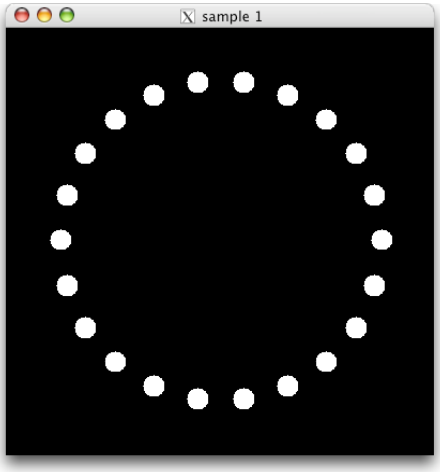
```
#include <stdio.h>
#include <math.h>
#include <eggx.h>

int main() {
  int x, win;
  float y, a, b;

  win=gopen(400,400);  /* 描画ウィンドウを開く */
  winname(win, "sample 1"); /* 名前をつける */
  newpen(win, 1);
  line(win, 5.0, 200.0, PENUP);
  line(win, 395.0, 200.0, PENDOWN);
  line(win, 200.0, 5.0, PENUP);
  line(win, 200.0, 395.0, PENDOWN);

  a=0.3;
  b=20.0;
  newpen(win, 2);
  for(x=-190; x<190; x++) {
    y=a * (float)x + b;
    pset(win, (float)x + 200.0, y + 200.0);
  }

  ggetch(win); /* キー入力を待つ */
  gclose(win); /* 描画ウィンドウを閉じる */
  return 0;
}
```

□ 課題 2.

三角関数など数学関数を利用して、何か美しい図形を描いて下さい。
例えば下記のようなものなど。各自工夫して、きれいなものを作って下さい。

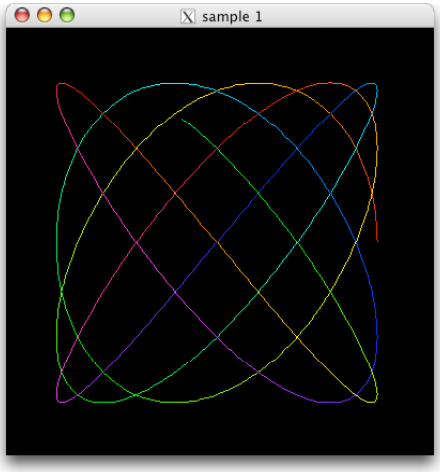


sin カーブを少しずらしながら
色を変えて描いたもの



円周上に中心位置をずらし
ながら図形を描いたもの

スパイラル



リサージュ曲線（描いている途中）
徐々に色を変えてアニメーションとしてみた