ゲノム情報科学研究教育機構　バイオインフォマティクス人材養成プログラム

プログラミング実習 11/18 13:00-17:00

Cプログラミング

担当(京都)：　瀧川 一学 (Ichigaku Takigawa)
takigawa@kuicr.kyoto-u.ac.jp

November 18, 2005 11:40 AM (by iWork '05 Pages)

概　要

C言語を知らない人を対象としてC言語の基礎について学ぶ。演習室のMac OS X上の統合開発環境Xcodeを用いる。

方　針

(1) 4時間の内容なので材料はC言語っぽいエッセンス**だけ**を扱う。省いたところが重要でないわけではない。

(2) セクション0をのぞき、基本的に**1セクションで1つだけ**短めのソースコードを提供し、この中身に素材を詰め込む。

(3) **ソースコードを見て何をやっているか考える。部分的に書き換えたりしていろいろ試してみながらC言語ぽさを体感する。**

(4) 演習で扱わない部分や扱った部分の細かい規則について興味を持った方は参考書籍などを適宜参照のこと。

以下はANSI C89規格の予約語(キーワード)一覧で**白いものだけ**を今回扱う。

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

目　次

参　考　書　籍

WebCT上のシラバスに書いた以外でいくつか参考にした書籍です。[1]は初心者向けにとても初歩的なところから丁寧な説明があります。[2]はもっとしっかりした本でひととおり書いてあります。より詳しく知りたい場合に使えます。[3]はポインタについて考察した本、[4]は古いけれどCの煩雑で不可思議な点に触れていて今でもわりとためになる有名な本です。[5]は実際のテクニックについていろいろ触れていて実践的なC言語の使われ方を知ることができます。

[1] 倉薫「プログラミング学習シリーズ C言語(1), (2)」翔泳社、2002.
[2] Les Hancock et. al.「改訂第３版 C言語入門」アスキー、1992.
[3] 前橋和弥「C言語 ポインタ完全制覇」技術評論社、2001.
[4] Peter van der Linden「エキスパート Cプログラミング」アスキー、1996.
[5] 多治見寿和「プログラミングテクニック UNIXコマンドのソースコードにみる実践的プログラミング手法」アスキー、2003.

前準備：Xcode及びCコンパイラ(cc)の使い方

1.Xcodeの起動

基本的には本演習ではXCodeという開発環境を用いることにします。

(1) デスクトップ上のシステムハードディスクをクリック

(2) Developerをクリック

(3) Applicationsをクリック

(4) Xcodeをクリック

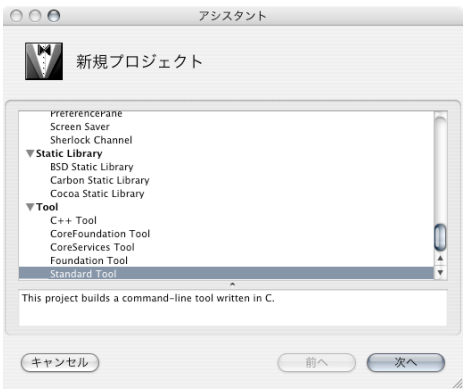
Xcodeがドックに入りメニューは表示されますがウィンドウは何もでません。

2. 演習用のプロジェクトの作成

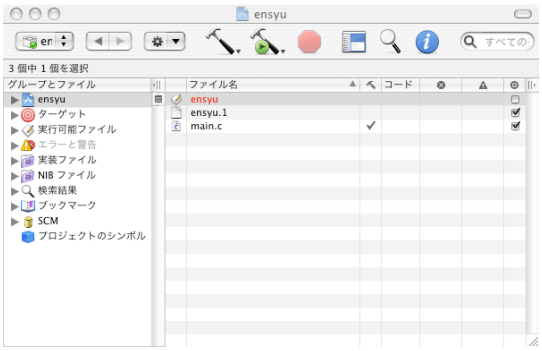
まず演習用にプロジェクトを新規で作ります。説明上は名前は「ensyu」にしますが、なんでも良いです。

上部メニューから「ファイル」→「新規プロジェクト」を選びます。

リストの一番下Tool → Standard Toolを選択し「次へ」をクリックします。



プロジェクト名に「ensyu」と入力し「完了」をクリックします。

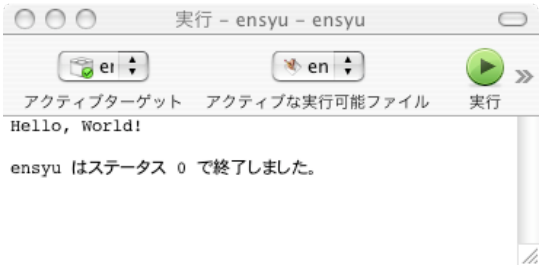


3. ソースコードの追加と編集

初期設定で「main.c」というファイルができているはずです。画面上のmain.cをダブルクリックして内容を見れます。これは通称「Hello World」と呼ばれる典型的な例です。

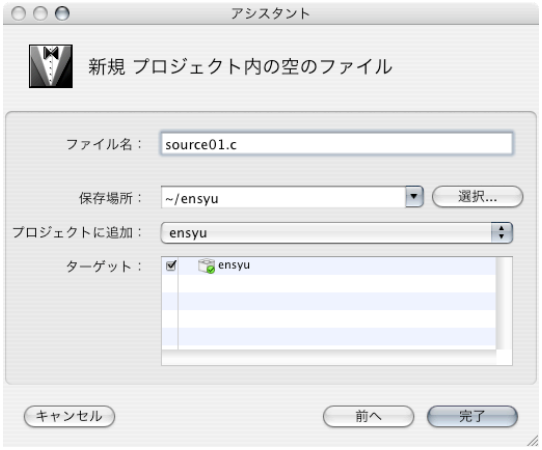
そのまま「ビルドして実行」をクリックするとビルド(機械命令への翻訳)されて実行されます。

「Hello Word!」という画面への出力と終了情報の文章が表示されます。



このままmain.cを書き換えても良いのですが例題ごとに新規ファイルを一から作ることにします。ensyuのSourceフォルダを選択した状態でファイルを追加します。

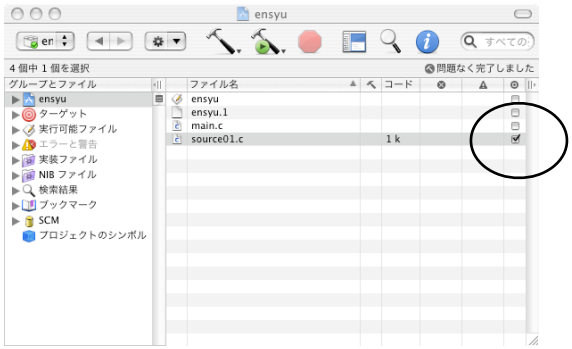
歯車のアイコンから追加→新規ファイルを選択。



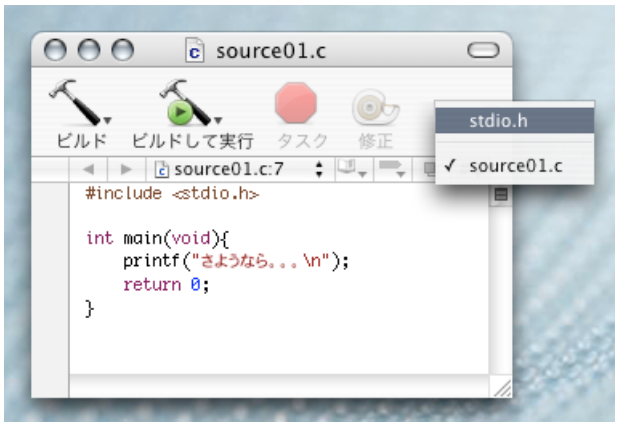
一番上の「プロジェクト内の空のファイル」を選びます。出てきたウィンドウのファイル名のところだけ変更します。ここでは「source01.c」としました。Cソースコード名には最後に「.c」をつけます。

空のウィンドウが開くのでそこに自分のプログラムを書き込みます。文法的に解釈され色がついたりします。インデント(行頭揃え)なども行われます。

ただし、ビルドする前にはターゲットにしたいソースコードだけにチェックがつくようにします。これをしないとエラーが出てしまいます。あとは「ビルドして実行」をクリックすることでビルドされてプログラムを動かすことができます。



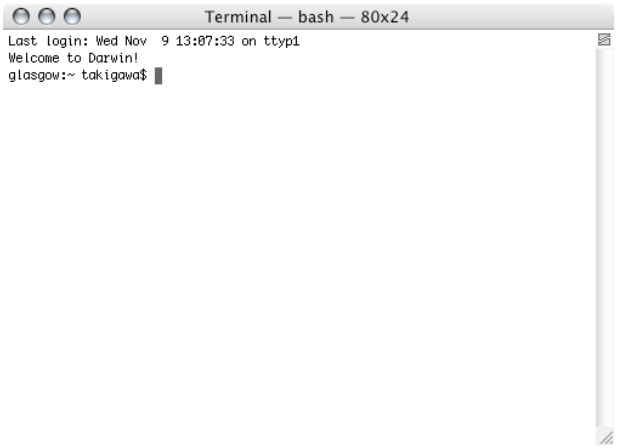
編集画面の右上の#ボタンからstdio.hを見ることもできます(使われたあとなら)。



課題ごとに上記の追加を繰り返せばプロジェクトひとつで複数の例題のファイルを管理できます。ただし、ビルドのときには、どのファイルをコンパイルするか(ターゲットをどのファイルにするか)を適宜切り替えて行ってください。

4. ターミナルの起動法

ドックに入っている右のアイコンか、システムハードディスク→アプリケーション→ユーティリティ→ターミナルをクリックしていき「ターミナル」を開きます。



この▮のところにキーボードから操作のためのコマンドを打ち込みます。

日本語を出力する場合、Control+クリック→「ウィンドウ設定」→「ディスプレイ」の「文字セットエンコーディング」を「Unicode (UTF-8)」などに変更する必要があるかもしれません。特にXcodeのエディタは初期設定として日本語にこれを用います。

5. Xcodeのソースコード

先ほど作成したXcodeのソースコードは自分でつけた名前「ensyu」というフォルダに格納されています。ドックアイコンの「Finder」から内容を見ることができます。

ここではここにあるソースコードをそのまま利用してCコンパイラ(ccコマンド)のコマンドラインからの利用法を示します。

6. ccの使い方

ccコマンドの実体はgccというプログラムです。ターミナルに「man gcc」と入力することでこのプログラムについての情報を参照することができます。ここではこれ以上の詳細には触れません。

ccコマンドを使って、Xcodeが初期生成したmain.cを機械命令に翻訳する手続きについて述べます。

結論から言うと「cc main.c」と入力すると「a.out」という名前で実行可能コードを得ることができます[1]。

```
Last login: Wed Nov  9 13:26:42 on ttyp1
Welcome to Darwin!
$ cd ensyu
$ ls -1G
build
ensyu.1
ensyu.xcode
main.c
source01.c
$ cc main.c
$ ls -1G
a.out
build
ensyu.1
ensyu.xcode
main.c
source01.c
$ ./a.out
Hello World!
$ hexdump a.out | head
0000000 feed face 0000 0012 0000 0000 0000 0002
0000010 0000 000a 0000 0500 0000 0085 0000 0001
0000020 0000 0038 5f5f 5041 4745 5a45 524f 0000
0000030 0000 0000 0000 0000 0000 1000 0000 0000
0000040 0000 0000 0000 0000 0000 0000 0000 0000
0000050 0000 0004 0000 0001 0000 018c 5f5f 5445
0000060 5854 0000 0000 0000 0000 0000 0000 1000
0000070 0000 1000 0000 0000 0000 1000 0000 0007
0000080 0000 0005 0000 0005 0000 0000 5f5f 7465
0000090 7874 0000 0000 0000 0000 0000 5f5f 5445
$
```

[1] hexdumpは0/1の並びになっているコンピュータ上のファイル内容そのまま(ダンプ)を16進数で表示するプログラムです。より古いものにodというコマンドもあり利用できます。odは8進数でのダンプです。一番左の数字は何バイト目の表示かという数字になっています(この数字自体も16進数表示になっています)。

7. コンパイル関連の処理

UNIX演習などに出席し、ターミナルでコマンドがある程度扱えるような方などは、以下のようなことも試してみると良いでしょう。

```
$ hexdump -cx main.c
```

main.c(テキストファイル)自体のバイナリダンプを見る。-cは文字があれば同時に表示する。

```
$ cc -E main.c | less
```

-Eオプションはmain.cの前処理だけを行う。#include文などが展開される。

```
$ cc -S main.c; less main.s
```

-Sオプションは前処理及びコンパイルを行う。main.cに対して、main.sというアセンブリ言語のコードを出力する。

```
$ as main.s
```

asコマンドはアセンブラ。上記で出力したアセンブリ言語のコード main.s をアセンブルする。ただし標準ライブラリのリンクはしない。出力は「a.out」。a.outという名前はassembler outputから来ている。

ただしこの段階のa.outはライブラリがリンクされていないので実行できない。

```
$ cc -c main.c; hexdump main.o
$ diff main.o a.out
$ cc main.c
$ diff main.o a.out
```

-cオプションはアセンブルまで行って止める。main.cに対してmain.oという出力コードが得られる。これは上記で生成したmain.sのアセンブル後のオブジェクトコードのa.outと同じ(二つのファイルの違いを取るdiffコマンドで確認)だが、「cc main.c」とやって生成されるa.outとは別物である(最後の行は差分が表示される)。

```
$ cc main.o -o my_exe; ./my_exe
```

オブジェクトコードを実行可能コードに変換する。リンカプログラム自体はldコマンドとして、MacにもあるがMacの実行可能コードの形式が少々複雑なせいか、単純に標準ライブラリ(libc.dylib)だけをオブジェクトコードにくっつけるだけではすまないようなので、ここでは詳細に触れないことにします[2]。

8. エディタ

Cプログラムのソースコードはテキストでc言語の仕様にそった命令が書いてあればどんなアプリケーションで作ってもかまいません。

Mac上ではたとえばシステムハードディスク→アプリケーション→テキストエディットで起動できる「テキストエディット」が挙げられます。

作業中のターミナルからこのアプリケーションを起動する場合は

```
$ open -a TextEdit
```

などとします。

また、Mac OS Xでは、UNIXオペレーティングシステムなどでよく使われるemacs, vi, picoなどのエディタもターミナル中で利用することができます。

また、XCodeでプロジェクトを作成せずに、メニューから、ファイル→空の新規ファイルとしてエディタだけ起動することもできます。ただし、この場合、ビルドボタンやC言語用の色づけなどは機能しないので気をつけてください。

9. bcコマンド

後で２進数、１０進数、１６進数などを変換できるようなツールが必要になると思います。bcコマンドのibase(入力が何進数か)とobase(出力が何進数か)を入れればこれを得ることができます。

```
$ bc
ibase=16
obase=2
FFA93
1111111101010010011
quit
$
```

[2] ちなみにlddコマンドはMacにはなく、オブジェクトコードの情報はotool、ライブラリにはlibtoolを使う。

00 C言語とは

歴史と特徴

❖ C言語

C言語は1972年頃ベル研究所でDennis Ritchieによって主にUNIXオペレーティングシステムを書くために開発されたプログラミング言語です[3]。

その後の様々な言語に影響を与えただけではなく、現在も組込みや主流なソフトウェア開発の言語として使われています。

❖ 系譜

Algol 60 [1960年、国際委員会]

FORTRANより数年あとに登場した洗練された言語で多くのプログラミング言語がAlgolの影響を受けている。抽象的で汎用的すぎ主流にならなかった。

↓

CPL [1963年、ケンブリッジ大 & ロンドン大]

Combined Programming Language。Algolを使いやすくし現実的な問題を解けるものを目指した。

↓

BCPL [1967年、ケンブリッジ大、Martin Richards]

Basic Combined Programming Language。CPLは機能が多すぎ覚えるのが難しく使いこなしづらかったので好ましい機能だけを選りすぐったもの。

↓

B [1970年、 ベル研、 Ken Thompson]

初期のUNIX用に設計された言語。機能を節約しすぎ、特定の問題には便利だが応用範囲が限られていた。

↓

C [1972年、ベル研、Dennis Ritchie]

↓

C++ [1983年、AT&Tベル研、Bjarne Stroustrup]

❖ 特徴

一人の人間によって設計された言語は設計者の専門分野で威力を発揮します。Cの場合、システムソフトウェアです。AlgolやFORTRANは経理処理や科学技術計算には向いていましたがシステムレベルの作業にはあまりに抽象的すぎて、OSやプログラミング言語を書こうとなるとやはりアセンブリ言語に頼らざるを得ませんでした。

そのような背景から設計されたBCPLやBは機械に密着しすぎて他のことにはたいして役に立ちませんでした。Cはこの中間の考えで作られシステムレベルの作業がやりたい放題できるわりに科学計算や経理処理など他のことにも使えるような汎用性の高さを持っています。

これは主に言語自体を小さくコンパクトに設計して機械依存しそうな部分を(C言語自身によって書かれた)ライブラリとして言語外に放り投げることでもたらされました。例えば、C言語自体には画面に文字を出す機能さえありません(標準入出力ライブラリによって提供されます)。

❖ 標準化

1989 **(C89)** ANSI X3.159-1989
1990 ISO/IEC 9899 : 1990(E)
1993 JIS X3010-1993 (上記の翻訳)
1999 **(C99)** ISO/IEC 9899 : 1999(E)
2003 JIS X3010-2003 (上記の翻訳)

---

[3] ただしUNIX自体はCより先の1970年にまずアセンブリ言語によって書かれました。UNIXの時間情報が1970年1月1日からの通算秒で保持されているのもこのためです。

❖ 学ぶ上での注意点

しかし、C言語はもともと**自分たちで使うために**作られ、ユーザのニーズにしたがって**かなり行き当たりばったりに**拡張されてきた**30年以上前の古い言語**で現在の言語と比べるとたくさん**理不尽な**点もあります。C言語はそれまでの巨大な仕様の言語が失敗してきた経緯から keep it small and simple を信条に作られましたが、その際に優先されたのは

♦ コンパイラを簡単に作れること
♦ 高速な実行コードを吐けること

など当時の需要であり、 現在求められる

♦ プログラムの読みやすさ・書きやすさ
♦ 違反検出や例外処理などの安全性確保
♦ コンパイラによる最適化

などはわりと無視されました。

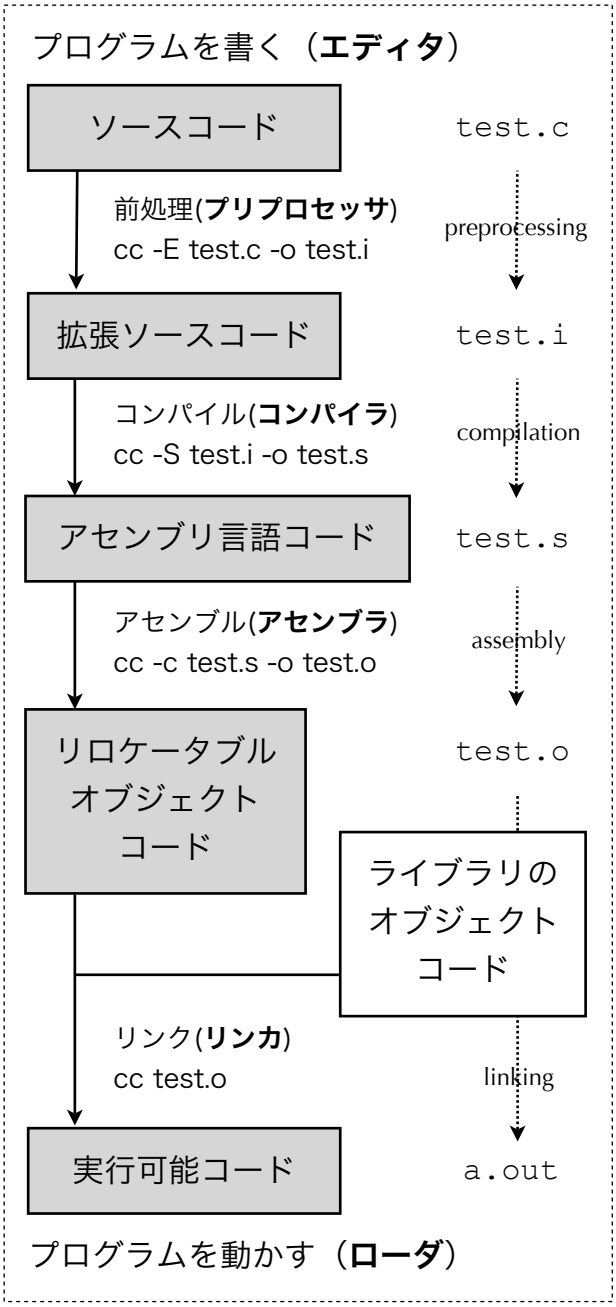
これらの問題が認識されるようになったころには今さら改変などできないくらいに広まってしまっており、どうにもこうにもいかない状況になっていたのでした。

なので、**変態的な文法や記法**とか**理不尽な例外**とかが混入していて、それがC言語の習得を難しくしてしまっているようです。C言語の思想自体は単純で奇麗なものでしたが、歴史的事情で固まってしまった標準仕様は**行き当たりばったりなままで変なもの**が多いので合理性とかをあんまり求めてはいけません☹。

このへんの**大人の事情を寛大な心でうけとめる**ことによって、現在まで30年以上に渡って世界を席巻しつづけるC言語の強力なパワーを享受できますのでそこいらへんはよろしくお願いします(?)。

❖ プログラムを動かす

エディタなどを使って自分で書いたC言語の命令を並べたテキストファイル(ソースコード)をそれに相当する機械レベルの命令(実行可能コード)に翻訳するには一般的に以下の手順に従います[4]。



普通は「cc test.c」だけで「a.out」ができるのでこうした細部は意識する必要はあまりありません。

[4] ただしコマンドccのオプションは演習で用いるAppleのgccのもの。a.outはassembler outputから来ているが実際はリンカの出力である。開発当時の計算機にリンカがなく、その名前だけが今も残っているとのこと。

01 関数と出力

プログラムは関数の寄せ集めで出来ている

```
/* Hell is filled
                  with amateur musicians.  */
#include <stdio.h>

void hello_goodbye(void);

int main(void){
   hello_goodbye();
   return 0;
}

void hello_goodbye(void){
   puts("こんにちは。");
   puts("でも、さようなら。");
}
```

❖ はじめに

ソースコードは半角英数で書いてください。テキストファイルであれば何で書いても良いです。

プログラムは最初の行から順に解釈されます。スペースや改行などはどう書いてもいいのですがそろえたほうがきっと読みやすいと思います。

上例のhello_goodbyeとかputsとか、C言語で使う名前には英数字の他アンダースコア(_)も使えます。ただし、名前は数字から始まってはダメです。大文字と小文字は区別され違うものと見なされます。

❖ コメント文

記号「/*」と「*/」に書かれた文章は**コメント**と呼ばれます。コンパイラはこの間に書かれた文章(改行も含めて)を見ません。したがって、何が書いてあっても良いですが、普通はプログラムが何をしているか補足するメモを書きましょう(例は悪い例です)。あとで見返したとき理解の助けになります[5]。

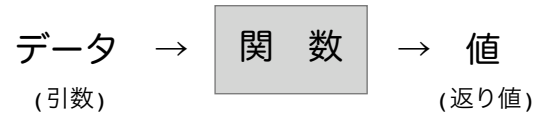
❖ #include文

#include <stdio.h>という記述はstdio.hというファイルの内容を#include文のところへ書き写せ、という**プリプロセッサに対する**命令です。 なぜこんな文が必要かは後述します。

❖ 関数

プログラムは**関数**の寄せ集めで構成されます。関数とはある命令文を集めた単位で「データを受け取って何らかの処理を行い値を出すもの」です。

[5] 書いているときはコメントなしでも余裕で分かっていても数日ほったらかしておくと書いた本人でさえ何をやっていたか、とんと分からなくなるものです。



上のプログラムではhello_goodbyeとmainとputsと３つの関数を使います。

MacやWindowsでCプログラムを動かす時にはまずmain関数から処理が行われる(関数が呼ばれる、と言う)と決まっています。このmain関数は仕様では

```
int main(void);
int main(int argc, char **argv);
```

のどちらかとなっています。

❖ プロトタイプ

上のmain関数以外に使う関数は**mainに先立って**どんなふうかそれぞれの説明(**プロトタイプ**)を書いておく必要があります。上の例ですと関数hello_goodbyeを使うために次のような文

```
void hello_goodbye(void);
```

を書いておきます。hello_goodbye関数の内容自体はmain関数より後でも先でもかまいません。

voidとintというのは「関数がもらうデータ(**引数**)」と「関数が出力する値」の種類に関する情報です。voidは何も無しということでintは整数ということです(02で説明しますが「**型**」と言います)。

```
出力値の型　関数の名前(引数の型);
```

がプロトタイプの記述になります。

main関数は整数0をreturn文で出力しています(値を返す、と言う)。この値はプログラムを実行したオペレーティングシステム自体に引き渡されます。0は正常終了、それ以外の値なら何か問題があって止まってしまった(異常終了)ことを表します[6]。

❖ ライブラリ

一方、puts関数のプロトタイプも必要なのですが、これは#includeしたstdio.hの中にもう書いてあるので要りません。

つまり、このstdio.hという名前のファイル[7]の中にはここで使われている文字を画面に出すための関数(putsなど)のプロトタイプなども書かれています(中身を見て下さい)。C言語は最も基本的な入出力にも**ライブラリ**(よく使うであろうということで既に用意されている関数集)を使うので何も#includeしないと画面に文字を出すだけですら大変なのです。

ライブラリの関数の内容自体はあらかじめ用意されており、その部分はリンカが最後にくっつけます。ANSIのC89規格では以下は**標準ライブラリ**として用意するように決まっています。

ヘッダファイル	内容
assert.h	診断機能
ctype.h	文字操作
errno.h	エラー
float.h	浮動小数点型の特性
limits.h	整数型の大きさ
locale.h	文化圏固有操作
math.h	数学
setjmp.h	非局所分岐
signal.h	シグナル操作
stdarg.h	可変個数の実引数
stddef.h	共通の定義
stdio.h	入出力
stdlib.h	一般ユーティリティ
string.h	文字列操作
time.h	日付および時間

[6] 例えば、OSにエラーコードを返したり、スクリプトで外部コマンドが正常に走ったか検査するときなどに使えます。

[7] “stdio”はstandard input/output (標準入出力)という意味です。

02 型と宣言と変数

変数を使う前にその型と名前を宣言しなくてはならない

```c
#include <stdio.h>

int plus(int a, int b);
int cube(int a);

int main (void) {
    int i1 = -3; int i2; int i3;
    char c1;
    double d1;
    i2 = i1 * 9;  i2 = i2 + 1;
    i3 = plus(cube(i1),i2);
    c1 = 'a';
    d1 = 3.14;
    printf("i1=%d,i1*i1*i1+i2=%d,i3=%d\n"
           "c1=%c(%d),d1=%f\n",
           i1,i1*i1*i1+i2,i3,
           c1,c1,(d1*4.0+1.0)/3.0);
    return 0;
}

int plus(int a, int b) { return a+b; }
int cube(int a) { return a*a*a; }
```

❖ 変数の値と型

変数とは「ある定められた範囲の値を取る数」のことです。変数は名前をつけて管理します。実行可能プログラムになったら名前の情報はもはや**保持されてません**のでどういう名前を付けても良いのですが、分かりやすいものを付けたほうが良いです。それぞれの変数は格納する「**値**」と、それがどんな種類のものかを示す「**型**」を属性として持っています。変数を使うには**使用に先立って**それに関する説明(**宣言**)を書く必要があります[8]。

```
int a;    double d;
```

などはint型で名前がaの変数、double型で名前がdの変数を使うという宣言です。

この演習ではint(整数)、double(実数)、char(文字)の型を使います。また、関数のプロトタイプや内容の引数にも型が設定されていることに注意してみてください。

```
int plus(int a, int b);
```

は二つのint型の引数を取ります。

❖ 代入文

宣言した変数には値を格納することができます。この操作を**代入**と言います。

```
c = 'a';    d = 3.14;
```

[8] 関数のプロトタイプと同じです。関数もmain中での使用に先立って宣言が必要でした。なぜ変数を使用に先立ってわざわざ宣言しておく必要があるのかは03にて説明します。簡単に言うと機械の都合です。

などが代入です。'a'は「a」という文字ひとつを表します。また、変数や数字は足したり(+)引いたり(-)掛けたり(*)割ったり(/)余りを求めたり(%)できます。これらの算術演算子とその優先順位については巻末の「C言語演算子の優先順位と結合規則」の表を参照してください。

❖ 代入記号「=」の注意

「=」記号は**数学における等号とは違い**、右の**値**を左の**変数**につっこむ、という意味しかありません。従って、

```
a = 1;  a = a + 1;
```

などということも可能です。これは最初に値1を変数aに入れたあと値a+1(=2)を変数aに入れ直します(最終的には変数aが持つ値は2になります)。また、

```
1 = 3 - 2;
1 + 2 = 3;
a + b = 3;
```

などは左辺が変数の名前でないので**ダメ**です。記号「=」は**数学のとは違います**！！

❖ printf関数

プログラムを実行すると次の結果

```
i1=-3,i1*i1*i1+i2=-53,i3=-53
c1=a(97),d1=4.520000
```

を画面に出力します。これは標準ライブラリのprintf関数が出力したものです。

printf関数はメジャーな関数のわりには複雑でしかも**可変引数(引数の個数が固定ではない)関数**です。基本的には書式を指定する文字列の「%とそのあとの文字」のところで変数の値を**文字として**出します。

```
int x=13, y=22;   printf("あい%dうえ%dお", x, y);
```

→ 出力：あいうえ13お　　変換文字は置き換え

以下はint x=789およびdouble y=3.14についてprintf(書式,x,y)が出力する例です。

書式	出力
"x=%d, y=%f"	x=789, y=3.140000
"x=%5d, y=%10f"	x= 789, y= 3.140000
"x=%05d, y=%07.3f"	x=00789, y=003.140

以下は%dのような変換文字の例です。

変換文字	意味
%d or %i	10進数 (decimal/integer)
%f	浮動小数 (float)
%p	アドレス (pointer?)
%u	符号なし10進数 (unsigned)
%x	16進数 (hexadecimal)
%o	8進数 (octal)
%c	文字 (character)
%s	文字列 (string)

❖ 文字列リテラル

printf関数の書式は""で挟まれた文字の並びである**文字列リテラル(文字列定数)**によって指定します。文字列リテラルは並べるとくっつきます。「"abc" "def"」は「"abcdef"」と同じになります。

❖ エスケープシーケンス

「\n」のようにバックスラッシュ記号から始まる文字は特殊文字として扱われます。「\n」は改行を意味します。ちなみに**Macでは「\」記号は「alt」+「¥」で入力**します。printfとエスケープシーケンスについてより詳しい情報はターミナルでコマンド「man printf」によって得られますので参考にしてください。

03 記憶クラスとスコープ

変数も関数もすべて結局のところメモリの上に確保される

```c
#include <stdio.h>

void incr1(void);
void incr2(void);

void incr1(void){
     int x = 77;
     x = x + 1;
     printf("incr1: %d\n", x);
}

void incr2(void){
     static int x = 077;
     x = x + 1;
     printf("incr2: %d\n", x);
}

int x = 0x01;

int main(void){
     extern int x;
     {
          register int x;
          {
            x = 0x0a;
            {
                 printf("main[%d]: %d\n", __LINE__, x);
                 auto int x = 0xfd;
                 printf("main[%d]: %d\n", __LINE__, x);
                 x = 0;
             }
          }
         printf("main[%d]: %d\n", __LINE__, x);
          x = 0;
      }
     printf("main[%d]: %d\n", __LINE__, x);

     incr1(); incr1(); incr1();
     incr2(); incr2(); incr2();

     return 0;
}
```

❖ 結果

変数の名前が全部 x です。このプログラムの実行結果と内容を見比べて何が起きたか推測してみましょう[9]。

❖ 記憶クラス

変数は「名前」と「型」と「値」以外に自動(auto)、レジスタ(register)、静的(static)、外部(external)の4種類どれかの**「記憶クラス」**を持ちます。

[9] __LINE__はデバッグ用の定数でプリプロセッサによって行番号に置き換わります。このようなデバッグ定数には他に__FILE__(ファイル名)、__DATE__(日付)、__TIME__(時刻)などがあります。

外部変数とは関数の外で定義されていてプログラム中のどの関数からでも(同一ソースコード中になくても)利用できる変数です。しかし滅多に使いません。

❖ 変数とスコープ

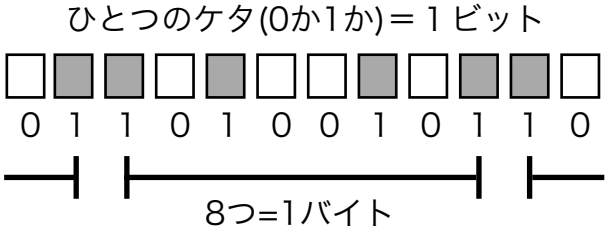
残り３つ自動変数、静的変数、レジスタ変数については**スコープ**という概念が重要です。スコープとは変数が生きている範囲のことで、該当変数が含まれる「{」から「}」までのブロックの中の**宣言文から「}」まで**のことです。この間でだけ変数を使うことができます。

autoは自動変数を示しています。ただし普通はautoとわざわざ書かなくても自動変数になるのでautoとは書きません。静的変数はstaticと書きます。この二つのスコープは同じですが、静的変数は関数が終了しても値を保持したまま、という点が違います(生き返ったときの宣言時初期化は無視され[10]、残っている値を使います)。

register変数は基本的には自動変数と同じ性質ですがメモリより速い記憶場所が利用できる場合(CPUレジスタなど)利用しようと試みますので変数へのアクセスが速くなるかもしれません[11]。

❖ 変数の実体とn進数

コンピュータ上では値はどんな型であれすべて**0と1**で表現されます。値はメモリという記憶場所に保存しますが、物理的にはおおざっぱに言って、**電圧がある/ない**が読み取れる小さい素子がたくさん順番にならんだようなものです。



次のようにsizeofに型の名前かその型の変数名を与えることで何バイト使うか調べることができます。

```
printf("char:%lu, int:%lu, double:%lu",
sizeof(char),sizeof(int),sizeof(double));
```

使用するコンピュータにも依りますが今のパソコンではcharは１バイト、intは４バイト(32ビット）、doubleはintの倍の８バイトであることが多いです。すなわち例えばコンピュータでは整数(int)は32個の0,1の並びで表現されていると言うことになります。

型の情報がないと何バイト見て良いかわからないので変数には型の情報とそれを使いますという宣言が必要です。

普通私たちが見る数字は「0〜9」の10種類の記号が一列に並んだものです。9まで行ったらケタが繰り上がって1からはじめます。「0と１」の２種類の記号でも同じようにできます。「０〜７」の８種類の記号や「０〜９、a〜f」の１６種類の記号でも同じように繰り上がりを考えてケタとして一列に並べていけば自然数を表すことができます。n種類の記号を並べて数を表現したものを**n進数表現**と言います。

C言語では数字の前に0をつけると8進数、0xをつけると16進数になります。準備の章の**bcコマンド**の説明を読み、n進数→m進数の変換を試してみましょう。

---

[10] 前述したように、宣言時の初期化は**単なる代入文とは違う**ので、混乱のもとになります。注意しましょう。

[11] 空いてなかったら普通の自動変数として扱われます。頻繁にアクセスされる変数のみに使用するだけのものです。

04 共用体・構造体とビット演算

要はメモリに並んでいる０と１の列をどう扱うかである

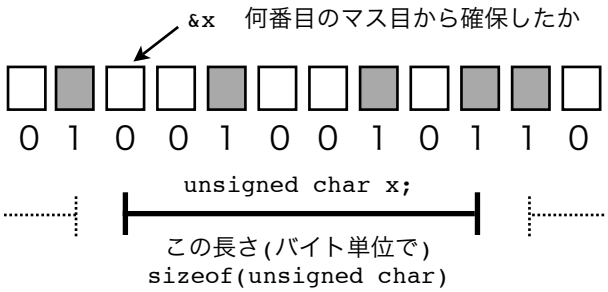
```
#include <stdio.h>

struct eachbyte {
     unsigned char c1;
     unsigned char c2;
     unsigned char c3;
     unsigned char c4;
};

union memory {
     int    i;
     double d;
     struct eachbyte byte;
};

void print_binary_digits(unsigned char c);
void print_memory(union memory mem);

int main(void){
     union memory mem;

     mem.i = 0567;
     print_memory(mem);

     return 0;
}

void print_memory(union memory mem){
     printf("mem.i    アドレス%pから%ldバイトぶん\n",&(mem.i),sizeof(mem.i));
     printf("mem.d    アドレス%pから%ldバイトぶん\n",&(mem.d),sizeof(mem.d));
     printf("mem.byte アドレス%pから%ldバイトぶん\n",&(mem.byte),sizeof(mem.byte));

     printf("hex -> %02X %02X %02X %02X\n",
             mem.byte.c1,mem.byte.c2,mem.byte.c3,mem.byte.c4);
     printf("bin -> ");
     print_binary_digits(mem.byte.c1);
     print_binary_digits(mem.byte.c2);
     print_binary_digits(mem.byte.c3);
     print_binary_digits(mem.byte.c4);
     printf("\n");
}

void print_binary_digits(unsigned char c){
     printf("%d",(c>>7)&1);
     printf("%d",(c>>6)&1);
     printf("%d",(c>>5)&1);
     printf("%d",(c>>4)&1);
     printf("%d",(c>>3)&1);
     printf("%d",(c>>2)&1);
     printf("%d",(c>>1)&1);
     printf("%d ",c&1);
}
```

❖ 結果

ここではさらに変数の中身がメモリ上でどうなっているかを詳しく見ます。この例では８進数の567がどうメモリ上に配置されるかを見ます。このプログラムを少し変えてできそうなことも試してみましょう。

❖ マス目の２色塗り絵

メモリ上の素子に電圧がかかっている(1)/かかっていない(0)かによってすべてを表現しなくてはなりません。この状況は並んでいるマス目(メモリ)を２色で塗る(0/1を割当てる)操作だけですべてを表現しなくてはならないことを意味します。少なくとも**どこからどこまで**のマス目にどの変数を置いたか分かるようになっていないとダメです。



上の図は x=0x25 (16進数で25) を確保したときの２色塗り絵(メモリ上の0/1)の様子です。

❖ アドレス演算子&

変数の名前の前に&をつけると何番目のマスから確保したか(先頭アドレス)を得ることができます。printfに%pで指定すれば、これを１６進数表現で見られます。

❖ unsigned型整数

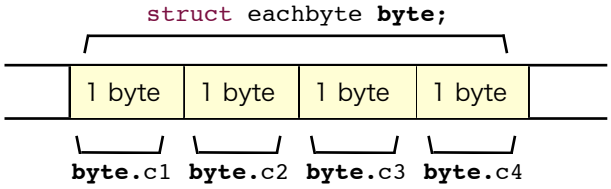
数字を0と1だけで表すとき負の数は問題です[12]。しかし正の数だけなら２進数として解釈すればすみます。unsignedというのを宣言のときにつけると正の数だけを取る変数を意味し、そのまま２進数で保持されるので、ここで用います。

それ以外の数の0/1表現はプログラムを書き換えて実際に見てみましょう。詳細は説明する時間がないので情報科学の本を参照してください。

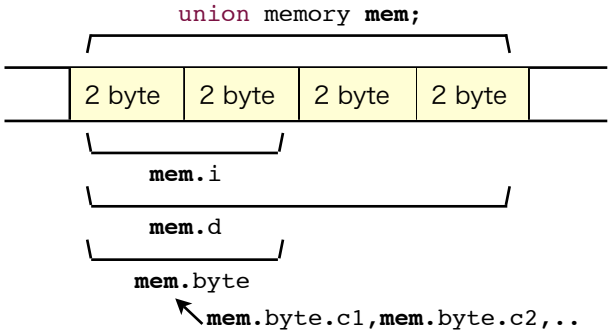
❖ 構造体と共用体

変数一つだけではなく複数の変数を並べて扱いたいことが良くあります。構造体structでは変数を順番にならべて配置します。共用体unionでは同じ０と１の並びをいろんな型の変数として取り出せるように配置します。

「struct eachbyte」の宣言ではunsigned charを４個順番に確保する(c1,c2,c3,c4)



「union memory」の宣言ではint,double,struct eachbyteを3個同じところから確保する(i,d,byte)



❖ 文字と整数のビット演算

char型やint型の値はマス目ごと(２進数ケタごと)に次のルールで演算できます。

反転　OR(和)　AND(積)　XOR

x	y	~x	x|y	x&y	x^y
0	0	1	0	0	0
1	0	0	1	0	1
0	1	1	1	0	1
1	1	0	1	1	0

例)

```
                    各ビットごとに演算
x=170     10101010
y=240     11110000      170&240
----------------
x & y     10100000      =160

x         10111010   [ビットシフト]
x>>1      01011101    x>>n はnビット右へ
x>>2      00101110   (x<<m はmビット左へ)
x>>3      00010111   0で埋めつつズラす。
```

[12] 「-」(マイナス)という記号も使えないということになります。２色塗り絵だけで何とかする必要があります。

05 制御構造

すべての手続きは逐次実行、分岐、反復、の３つの流れで書ける

```c
#include <stdio.h>

enum   _type  { Integer, Real };
union  _value { int i; double d; };
struct number {
     enum  _type  type;
     union _value value;
};

void print_binary_digits(struct number n);

int main(void){
     struct number n;

     n.type = Integer;
     n.value.i = -10;
     while(n.value.i <= 10){
          printf("%3d => ", n.value.i);
          print_binary_digits(n);
          ++n.value.i;
     }

     n.type    = Real;
     n.value.d = 3.14;
     print_binary_digits(n);

     return 0;
}

void print_binary_digits(struct number n){
     if(n.type != Integer){
          printf("整数じゃないです。");
     }else{
          int i = 8*sizeof(int)-1;
          while( i >= 0 ){
               printf("%d",(n.value.i>>i)&1);
               if( i%8==0 ){
                    printf(" ");
               }
               --i;
          }
          printf("\n");
       }
}
```

❖ 実行の流れ

C言語のソースコードは基本的に最初から順番に実行されていきます。printfを三回書いたら書いた順番に文字が出力されます。この逐次実行以外に、条件分岐と反復(ループ)を用いて実行の流れを書くことができます。

❖ 反復処理(while)

ある文を繰り返して実行する反復処理はwhile文として書けます。例えば、次の例

```
int i = 0;
while( i < 10 ){
    printf(”%d”, i);
    ++i;
}
```

は変数 i を0〜9まで変えて印字します。条件「i < 10」が満たされる間、ブロック{ }の中身を繰り返します。条件がいつまでも満たされているといつまでも繰り返します。このような処理は非常に良く現れるためC言語には次のfor文という専用の別記法があります(演習では使いません)。

```
int i;
for( i=0 ; i < 10 ; ++i ){
    printf(”%d”, i);
}
```

❖ 条件分岐(if)

ある条件が満たされる場合と満たされない場合で異なる処理をさせたい場合、if文(とelse)による条件分岐で書けます。

```
int i = 30;

if ( i > 20 ) {
   printf(”20より大きい”);
} else if ( i > 10 ) {
   printf(”20より小さいけど10より大きい”);
} else {
   printf(”20よりも10よりも小さい”);
}
```

2番目のブロックは変数 i の値が20より大きいときは実行されないことに注意してください。if文では満たされる条件によってブロック**どれか**が実行されます。

❖ 列挙型(enum)

取り得る値が有限個に決まっている場合、**enum型**という型でそれを指定できます。初期化時に取り得る値の名前を書いておきます[13]。

```
enum month = { Jan, Feb, Mar, Apr, May,
Jun, Jul, Aug, Sep, Oct, Nov, Dec };

enum month your_month(void);
enum month my_month;
    :
month = Feb;
```

上のように記述すると関数 your_month は月を返すんだ、とか変数 my_month には月を保持するんだとか分かりやすくなります。

ちなみに**名前の頭に_がついていることには構文上の意味はありません**。実際は、構造体の名前とメンバの名前と構造体変数の名前は別々に管理されるので

```
struct foo { int foo; };
struct foo foo;
```

も可能なのですが混乱すると思います。

❖ 関係演算子・論理演算子

反復にも条件分岐にも条件を論理的に記述することができます。例えば、「i > 20」は「変数 i が20より大きい」という論理式で真か偽か(条件が満たされるか、満たされないか)どちらかです。関係や論理には以下の記号を用います。

関係演算子

a == b	aがbと等しい	a != b	aがbと等しくない
a > b	aがbより大きい	a >= b	aがb以上
a < b	aがbより小さい	a <= b	aがb以下

論理演算子

x	xである	! x	xではない
x && y	xかつy (論理積)	x || y	xまたはy(論理和)

優先順位については巻末の「C言語演算子の優先順位と結合規則」の表を参照してください。

[13] プリプロセッサ命令の#define文で書くことのほうが多いようですがCプリプロセッサの命令についてはまたいろいろありますので今回は#include以外は扱いません。参考文献などを参照してください。

06 ポインタと動的メモリ割当

ポインタとは文字通りポイントするだけで実体は別である

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

void print_int(int arg);

int main(void) {
     int *pointer;
     int i;
     int n=10;

     pointer = malloc(sizeof(int)*n);

     i = 0;
     while(i < n){
          *(pointer+i) = 0xffff << i;
          ++i;
     }

     i = 0;
     while(i < n){
          printf("%p : ", pointer);
          print_int(*pointer);
          ++pointer;
          ++i;
     }

     free(pointer-10);
     return 0;
}

void print_int(int arg){
     int i = 8*sizeof(int)-1;
     while(i >= 0){
          printf("%d",(arg>>i)&1);
          if(i%8==0){  printf(" "); }
          --i;
     }
     printf("\n");
}
```

❖ ポインタ型

どのマスから並べたかという数字(アドレス)を取っておく変数の型を**ポインタ型**と言い、int, char, dobleなど、それぞれの型から派生して生じる特別な型です。数字を格納するだけならint型でいいじゃないか、なぜ型ごとに区別する必要があるのだ？と感じると思いますが、「何マス目から置いてあるか」だけ情報を得ても、「何マス使ったか」事前に分からないと情報の取り出しようがありません。そのため、格納してある変数の型の情報を併せて指定して宣言します。

❖ ポインタ変数

ポインタ型の変数を使うときは基本的には派生元の変数の型(被参照型)のあとに「*」を付けて宣言します。int型の変数の値をどのマスから置いたかを格納するポインタ型はint*型、char型の変数をポイントするポインタ変数ならばchar*型というふうになります。

❖ 混乱した記号＊

ただしポインタの記法は混乱しており、

```
int* pointer;
int *pointer;
int * pointer;
```

のどれでも宣言できます。ただし複数同時に「, 」で宣言するときには、

```
int* a, b, c;
```

とすると**ポインタ変数はaだけでbとcは単なるint型変数になる**ので、a,b,cともにポインタとして複数同時に宣言するには

```
int *a, *b, *c;
```

などとする必要があります。

これにはポインタ変数のポイント先の内容を得るとき「*変数名」[14]で得られ、int* xという宣言をしたあとは、「*xがint型として扱える」という見方もあると言います。が、どんな解釈をしても結局、一貫性は得られません。Cの**宣言文と実際に変数を違う文は構文が異なる**ことを常に注意するべきです。割り切りましょう。

ポインタは例えば次のように使います。

```
int x = 50;
int *pointer_to_x;

pointer_to_x = &x;
printf("%p : %i\n",
       pointer_to_x, *pointer_to_x);
```

❖ 動的メモリ割当(malloc)

でも、次はどれもエラーか警告になります。

✘ (1) `int *x = 90;`
→ 宣言文では右辺はint値ではダメ。
`int y=3; int *x=&y;` とかが正常。

✘ (2) `int *x;  *x = 90;`
→ x はまだどこもポイントしていない。

✘ (3) `int *x;  x = &90;`
→ &演算子は値でなく変数につける。

✘ (4) `int *x;  x = 90;`
→ xは int*型 だけど 90 はint型の値[15]
またこれはxを90というアドレスを指すという意味になり、意味が異なる。(1)も。

C言語では**型と変数と値を区別することが重要**ですがポインタの場合、**すべてポインタと呼ばれる**ことも混乱の原因です。まず、名前を付けるにせよ、名前を付けないにせよ、値90を格納する変数の領域をとる必要があります。

```
int *pointer;   int x;
x = 90;  pointer = &x;
```

と変数xを用意するか、malloc関数を使い

```
int *pointer;
pointer = malloc(sizeof(int));
*pointer = 90;
```

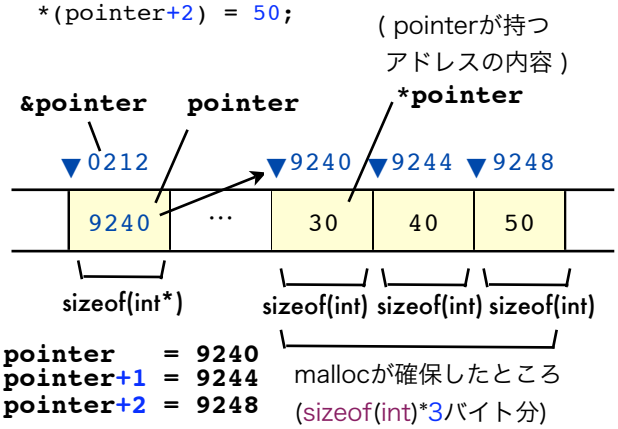
とポイントすべき領域を用意します。

malloc関数にはstdlib.hが必要で、 malloc(数字)で数字バイト分の領域(マス目)を用意し、どのアドレスから確保したか返します(malloc自体は型の情報は見ません)。領域を使い終わったらfree関数で使い終わったことを示しておきます。

❖ ポインタ演算

ポインタを利用してmallocで確保した領域を**同じ型の変数が順番に並んだもの**として使うことができます(intのサイズの3倍を3個のintで使う)。

```
int *pointer;
pointer = malloc(sizeof(int)*3);
*pointer = 30;
*(pointer+1) = 40;
*(pointer+2) = 50;
```



ポインタ変数は**ポインタ演算**と言う特殊な演算しかできません。ポインタ変数の加算は被参照型の大きさ単位で行われます。ポインタ変数は乗算など加算以外の演算はできません。

---

[14] このポインタ変数のポイントする内容を取りにいくための「*」は**間接演算子**と呼ばれます。

[15] 90というアドレスをxにセットしたい場合は後述の型変換(type cast)を行えば文法的に合法です。

07 関数とポインタ

ポインタは単に開始点をポイントしているだけである

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

void print_bytes(void* _pointer, int size);

struct dummy {
     int a;
     int b;
};

int main(void){
     int i;
     double d;
     struct dummy test;

     d = DBL_MAX;
     i = INT_MAX;
     test.a = 0xdead;
     test.b = 0xface;

     print_bytes(&d,sizeof(d));
     print_bytes(&i,sizeof(i));
     print_bytes(&test,sizeof(test));

     print_bytes(main,100);
     print_bytes(print_bytes,100);

     return 0;

};

void print_bytes(void* _pointer, int size){
     unsigned char *pointer = (unsigned char*) _pointer;
     int i=0;
     while(i<size){
          printf("%02x ", pointer[i]);
          ++i;
       }
     printf("<= from %p (%i byte)\n", pointer, size);
}
```

❖ ポインタを何に使うか？

「変数をポイントする」ことが何の役に立つのかと感じると思います。ポインタ操作によって実現できる便利なものとしては主に以下のようなものが考えられます。

(1) 返り値以外の形で値をもらう

(2) 関数や配列[16]を引数として関数に渡したり、返り値としてもらったりする

(3) ツリー、リスト、ハッシュ、可変長配列などの色々なデータ構造を表現する

[16] 配列については次節で扱います。

❖ 引数と値渡し

有名な例として次のswap関数は実際にはxとyの値を期待するようには入れ替えてくれません。

```
void swap(int a, int b){
    int temp;
    temp = a;   a = b;   b = temp;
}
    :
int x, y;
x = 99;  y = 193;
swap(x, y);
printf("x=%i, y=%i\n", x, y);
```

関数swap中の引数 a, bは**仮引数**と呼ばれ、スコープは関数内ブロックだけです。実際関数が呼ばれるswap(x,y)のx, yは**実引数**と呼ばれます。まず、a=x, b=yと実引数の**値が**コピーされます。a, bの実体はx, yとは別の領域に確保されます。C言語では関数へはこのような**値渡し**で引数が受け渡されます。ANSI C89以降は構造体・共用体も直接代入できるため値渡しです。

アドレス値を値渡し[17]すればポイント先の変数の内容を*演算子で取れますので関数によって変数を書き換えられます。

```
void swap(int* to_a, int* to_b){
     int temp;
     temp = *to_a;
     *to_a = *to_b;
     *to_b = temp;
}
     :
swap(&x, &y);
```

❖ 添字演算子[ ]

pがポインタ変数のとき*(p+1)や*(p+2)などの操作はポインタ演算が見づらいので

`*(p+i)` を `p[i]` と書く

という記法が用意されています。p+iとi+pはポインタ演算として同じアドレスを指すので
***(p+i)=*(i+p)=p[i]=i[p]** です。

この[ ]のような人間が読みやすくするためだけに用意された構文をしばしば**糖衣構文(Syntax Sugar)**などと呼びます。

❖ 書込み禁止領域

最近のオペレーティングシステムでは、関数や文字列リテラルなど書き換えるべきでないものはメモリ中の書込み禁止領域に置くため、現在は書き換えようとするとエラーになる事が多いです。

❖ 関数へのポインタ

関数へのポインタを宣言し通常のポインタ変数として使うことができます。**関数の名前は式中ではポインタに読み替えられます。**ちょっとファンキーな例ですが次のように関数自体を返すこともできます。

```
enum boolean { TRUE, FALSE };
int one(void) { return 1; }
int two(void) { return 2; }
int (*switcher(enum boolean flag))(void){
   if (flag == TRUE){ return one; }
   else{ return two; }
}
    :
int main (void) {
    int x, y;
    int (*func)(void);
    func = switcher(FALSE); x = func();
    func = switcher(TRUE);  y = func();
    printf("x=%i, y=%i\n", x, y);
    return 0;
}
```

宣言「`int (*func)(void);`」はポインタ変数funcがvoid型の引数をひとつ持ち、int型の値を返す関数をポイントするものである、という意味です。同様に、enum boolean型を引数にとる関数switcherは「void型の引数を持ちint型を返す関数へのポインタ」が返り値になります。

❖ 型キャストとvoid*型

型は「(型名)」を付けることで無理矢理変換する(**キャストする**)ことができます。実数から整数への型キャストなど値に変化が生じる場合もあります。int i = (double) 3.14; は i==3 になります。

void型変数は存在しません(void x;など)が「`void*`」という任意のポインタを型キャストなしで代入できる**総称的なポインタ型**が用意されています。例題ではこれを利用して

```
print_bytes((int*) 0xbffffd58,4);
```

のように任意の開始アドレスから任意バイトを印字する関数を定義しています。

---

[17] C言語にはC++のように変数のエイリアスとして機能するような参照渡しはありません。

08 配列とポインタ

配列は同じ型の変数を並べたものでポインタを使って操作できる

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

struct point {
     double x;
     double y;
};

int main(void){
     struct point data[10];
     struct point *pointer;
     int i;

     i=0;
     while(i<10){
          data[i].x = (double) random()/RAND_MAX;
          data[i].y = (double) random()/RAND_MAX;
          ++i;
     }

     pointer = data;
     i=0;
     while(i<10){
          printf("Point %d: x=%5f y=%5f\n", i, pointer->x, pointer->y);
          ++pointer;
          ++i;
     }

     return 0;
}
```

❖ 配列

配列とは一種類の型のデータを複数まとめて扱う仕組みです。今までは構造体のメンバで並べたり、mallocでメモリを割当てたりしてきました。配列として宣言することでmallocのような実行時ではなく最初から与えられた個数だけ並べることができます。これらはmalloc/freeのような明治的な領域確保は必要がなく、構造体や通常の自動変数のように自動的に確保・削除されます。

3個ぶんのint型の変数の並びを配列として用意したい場合

```
int array[3];
```

と宣言します。「array」がこの配列の名前になります。この宣言以降の式文の中では配列名arrayは常にこの並びの**先頭アドレスを指す定数ポインタとして読み替えられます。**

従って、array[i]や*(array+i) で i+1 番目に置いてある変数の値を取ったり、代入したりすることができます。これは糖衣構文として用意されている添字演算子[ ]によりarray[i]としても書き表すことができます。

```
&array[1]       or   &1[array]
array+1         or   1+array
&(*(array+1))   or  &(*(1+array))
```

は全て同じアドレスを指します。array[3]での**宣言**(要素が3個です)が、**使うとき**にはarray[**0**], array[1], array[2]まで(つまり*(array+**0**), *(array+1), *(array+2))であることに注意してください。array[3]や*(array+3)は確保した領域外を見

てしまいますので使えません(エラーになるかどうかは処理系やコンパイラによります)。

しかし、**定数**ポインタですので、arrayを int array[3]; で宣言したあとで、

✘ `array = malloc(sizeof(int)*3)`

などと保持している**先頭アドレス自体を書き換えることはできません**。従って、配列同士を直接配列名で代入したりも**できません**。array++などと書き換えることも**できません**。しかし、ポイント先の値自体は `array[0] = 5` または

```
*array = 5;
```

などとして書き換えられます。また、ポインタ変数へ配列名の定数ポインタを代入する操作は通常通りに可能です。

❖ 配列とポインタ

配列とポインタは本来全く違うのものです。配列というのは「同じ型の変数が複数並んだもの」であり、ポインタというのは「ある変数のアドレスを格納する変数」です。ここに混乱がしばしば生じるのは配列をポインタで操作するときのいくつかの特例的な構文なせいだと思われます。

(1) 宣言時初期化の構文
(2) 関数の引数でのポインタへの読替え
(3) 文字列(char型配列)に対する特例

❖ 宣言時の[ ]

配列を宣言する際に配列のサイズが確定できる状況では要素数を省略することができます。これは特殊な状況なので注意してください。

```
int array[3];
array[0]=0; array[1]=1; array[2]=2;
```

という代入は宣言文での初期化子によって

```
int array[3]  =  { 0, 1, 2 };
```

または

```
int array[]  = { 0, 1, 2 };
```

と書く事ができます。これは**宣言時の特殊な構文**であって式文の中では

✘ `int array[3];  array = { 0, 1, 2 };`

や

✘ `int array[3];  array[] = { 0, 1, 2 };`

などと代入することはできません(ただし次節で扱うように「文字列リテラル」が関係するchar型の配列の場合のみもうすこし複雑に見えます)。

しかし、ポインタにおいては宣言時に確保されるのは自身の領域(int*ならsizeof(int*)バイト)だけです。従って、ポインタ変数の宣言時に

✘ `int *array = { 0, 1, 2 };`

という初期化はできません(ただし、次節で見るようにchar*型つまり文字列の場合だけ例外的)。

❖ ポインタへのポインタ

二次元の配列は例えば int a[3][5];として宣言できます。このときも配列名aは先頭アドレスへの定数ポインタだが a[i][j] への参照は (a[i])[j] あるいは *((*(a+i))+j) という形で行われます。つまり、a[i]はint型へのポインタ変数になっています。このように二次元配列はポインタへのポインタとして操作することもできます。

❖ 関数の仮引数での特例

関数の仮引数の宣言においてだけはint *arrayとint array[ ]は全く同じようにポインタとして読替えられます。一般の宣言においてはポインタとして宣言することと配列として宣言することは**重大な差異**があります。後者の**配列名は**定数ポインタとして読み替えられますので、**書き換えはできません**。ポインタ変数は一般にポイント先自体も書き換えて使用されるものです。

❖ 例題に関する注意

stdlib.hを#includeすることでrandom()で0からRAND_MAXまでの整数乱数が得られます。また p->x は (*p).x の単なる糖衣構文です。

09 文字列リテラルと文字列処理

関数とコメント

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

char* complement(char* seq);

int main(void){
    int i, length;
    char *seq, *comp_seq;

    length = 20;
    seq = malloc(length+1);                  /* if (sizeof(char) == 1) */

    i = 0;
    while(i<length){
        seq[i++] = "atgc"[random()%4];
    }
    seq[i] = '\0';

    comp_seq = complement(seq);

    printf("starting   DNA: (5') %s (3')\n", seq);
    printf("complement DNA: (5') %s (3')\n", comp_seq);

    free(seq);
    free(comp_seq);

    return 0;
}

char* complement(char* seq){
    char* comp_seq;
    int i, j, length = strlen(seq);

    comp_seq = malloc(length+1);             /* if (sizeof(char) == 1) */
    strcpy(comp_seq, seq);

    i = 0;
    while(i<length){
        j = (length-1)-i;
        if     (seq[i]=='a'){ comp_seq[j]='t'; }
        else if(seq[i]=='t'){ comp_seq[j]='a'; }
        else if(seq[i]=='g'){ comp_seq[j]='c'; }
        else if(seq[i]=='c'){ comp_seq[j]='g'; }
        ++i;
    }

    return comp_seq;
}
```

❖ 文字列と文字配列

文字列リテラルは最後に終端文字 '\0'を持つ単なるchar型の配列なので式の中ではポインタに読み替えられます。しかし、配列として宣言するときとポインタ操作だけで扱うときは異なります。

```
(1) char *str;   str = "hello";
(2) char *str = "hello";
(3) char str[] = "hello";
(4) char str[10] = "hello";
```

は正しい初期化です。

"hello" == { 'h', 'e', 'l', 'l', 'o', '\0' }

は式文では上のchar配列へのポインタとして読み替えられますが、**配列の宣言文の中だけ**は初期化子として配列のまま**特別に解釈**されます。(1),(2)は**strというポインタ変数**を宣言しているのに対して、(3),(4)は**strという配列**を宣言しています。

✘ `char str[10];  str = "hello";`

✘
```
char str[10]; char *ptr="hello";
str = ptr;
```

はできません。ポインタ変数への代入、例えば、後者で `ptr = str;` は可能です。

❖ 文字列処理関数

詳しく扱えませんが文字列を操作する関数群はstring.hを#includeすることで利用できます。C言語では配列の大きさは自前で管理する必要がありますが、文字配列(文字列)だけは終端文字があるためにstrlenという関数で長さを取れます。またstrcpyは文字配列の内容のコピーに使います(=による代入は先に述べてきた理由で不可能なため)。

```
    char str1[] = "string";
    char *str2 = "string";

    strcpy(str1, "STRING");
/*  str1 = "STRING"; はダメ */
    str2 = "STRING";
/*  strcpy(str2, "STRING"); はダメ */
```

❖ 文字列遊び(おまけ)

以下が何を出力するか考えることで、以上の事柄をより具体的に理解できる。文字列と文字配列、ポインタ操作を考える。

```
    printf(&"abcdefg"[1]);
    printf("%c\n", "abcdefg"[1]);
    printf("%c\n", *("abcdefg"+2));
    printf("%s\n", "abcdefg"+3);
    printf("%d\n", 'z'-'a'+1);
```

これを用いて出力したいメッセージを不明瞭にしていくこともできます。添字アクセスa[i]はi[a]と書いても同じであったことに着目することによって、例えば、

```
printf("hello\n");
↓
printf(&2["so%s\n"],"hello");
↓
printf(&2["so%so\n"],"go to hell"+6);
↓
printf(&2["so%so" "\n"],
      "go to hell"+'g'-'a');
↓
printf(&2["so%so" "\n"],
       0["good."]+"go to hell"-'a');
```

などと意味不明にしていけます[18]。これに意味はありませんし、数字をさらに計算によって細工するなど不明瞭にしていくのは切りがないのでこのあたりで止めておきましょう。C言語の文字列の理解を深めるには良いかもしれません。

[18] The International Obfuscated C Code Contest (IOCCC) http://www.ioccc.org/

付録

C言語演算子の優先順位と結合規則

優先順位	演算子記号	意味	結合規則
高	( )	関数呼び出し	左から右
↑	[ ]	配列添字	左から右
|	.	ドット(構造体のメンバ)	左から右
|	->	矢印(構造体のメンバ)	左から右
|	!	論理否定	右から左
|	~	1の補数	右から左
|	-	単項マイナス	右から左
|	++	インクリメント	右から左
|	--	ディクリメント	右から左
|	&	アドレス	右から左
|	*	間接	右から左
|	(型)	キャスト	右から左
|	sizeof	サイズオブ	右から左
|	*	乗算	左から右
|	/	除算	左から右
|	%	剰余(モジュロ)	左から右
|	+	加算	左から右
|	-	減算	左から右
|	<<	左シフト	左から右
|	>>	右シフト	左から右
|	<	より小さい	左から右
|	<=	以下	左から右
|	>	より大きい	左から右
|	>=	以上	左から右
|	==	等しい	左から右
|	!=	等しくない	左から右
|	&	ビットごとのAND	左から右
|	^	ビットごとのXOR(排他的OR)	左から右
|	|	ビットごとのOR	左から右
|	&&	論理的AND	左から右
|	||	論理的OR	左から右
|	? :	条件(3項演算子)	右から左
↓	=	代入	右から左
低	,	カンマ演算子	左から右

メモ欄

1 byte = 8 bit

2^8 = 256

10進数(decimal)	2進数(binary)	16進数(hexadecimal)
000	0000 0000	00
255	1111 1111	FF
016	0001 0000	10
100	0110 0100	64
015	0000 1111	0F
051	0011 0011	33

10進 - 2進 - 16進　対応表

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

4ケタの２進数では 0〜15 まで (1ケタの16進数と同じ)

２ケタの16進数では 0〜255 まで (ちょうど8bit=1byteで表現できる整数と同じ)