cool-da

だってクールだ　C　グロサリー

c-glossary

グロサリー１　関数、ファンクション、**function**

関数とファンクションは同じ意味、functionの訳語が関数となります。
コンピューター以前に、算数方面でのfunctionの訳語として、関数、
という言葉がすでに存在していました。
それを、そのまま流用しているので、イメージがわきにくいと思います。
プログラムの裏側では、関数というものを見た時、どんな処理が行われているのか、
Cの関数の実像にググッと迫ります。
最初に、Cの関数の機能を調べます。

1. 関数には、プログラムの実行の順序を変更する機能がある。
2. 関数内で、新しくデータを作った場合の為に、使い回しが簡単にできる、スタック
   というメモリーの仕組みを利用する。これにより、プログラマーは、
   メモリーの使用量に対して、ある程度、目星をつけられる。

プログラムの実行の順序の変更に際して、そこで利用する専用の使い回しメモリーを差し出す、
というユニークなアイデアは、メモリーが極端に少ない時代での、いわば窮余の策でしたが、
現在も変わらずに使われています。
これらは、実際にどのように運用されているのでしょうか、舞台裏の様子をもう少し見ていきます。
Cは、文字ベースの変換プログラムで、わたしたちが、テキストエディターで書き込んだ、
cool.cファイルを、cool.oファイルに変換します。
つまり、Cの書式で記入したものを、アセンブリー言語に翻訳するというわけです。
アセンブリー言語のプログラムは、文中にて、特別な指定をしなければ、
上から下に、一気に実行されます。
Cは、わたしたちが書いたスーパープログラム、cool.cを見て行きますが、
関数を見つけたら、ジャンプ命令、jmpを加えます。
つまり関数の領域は飛び越してしまうような指定が入ります。
なぜでしょうか。

関数は、本体と呼び出しの２本立て。

関数は、本体と呼び出しの２本立てになっているからです。
呼び出しに遭遇したら、今度は、本体の入り口に向けて、ジャンプします。
本体の実行が済んだら、先ほど呼び出しをかけた行番号の、次の行に、
ジャンプする指令が入ります。
これで、プログラムは、もとの順番に沿って実行されます。
ここまでの説明によれば、Cの関数とは、
ジャンプ命令を３回使ったプログラムの流れの変更だ、とも見て取れます。
Cは、これは関数本体だ、というのをどうやって見つけるのでしょうか。
たとえばルビーでは、defという予約語を使います。
本体は、

```
def  myfun
   puts “Hello”
end
```

呼び出しは、

```
myfun
```

Cには、関数のはじまりを告げる予約語はありません。
関数の中で計算などの作業を行った結果を、呼び出しサイドに返す仕組み、がありますが、
この結果として返すデータの型の明示を、関数の先頭で行います。
もし、結果を戻す予定がない場合は、空を示すvoid、を書き込みます。
関数の中で操作するデータを、呼び出しの時に渡したい場合、
関数の名前の後に括弧をつけ、そこに書き込みます。
渡すデータが無い場合でも、括弧だけは書いておきます。
ルビーのように省略はできません。

Cの関数本体

```
void myfun() {
   puts(“Hello”);
}
```

呼び出し

```
myfun();
```

Cの関数の書式の特徴は。

Cの関数の書式の特徴は、本体では、

1. 関数本体の先頭には、戻り値の型を記入する。無しの場合は、voidと記入。
2. 名前の後には、かっこが必須。
   渡すべきデータがあれば、括弧のなかに、型情報と名前を書いておく。
3. 関数の領域を示すものとして、波かっこ（カーリーブレイス）を使う。
4. 関数の中には、ゼロ個以上の文を置きます。

一方、関数の呼び出し文では、

1. 関数を呼び出し、本体が実行されると、制御は呼び出しの次の行に必ず戻ってくる。
2. 舞台裏では、スタックが働いている。
3. 関数の呼び出しは、関数の中から行う必要がある。

関数の呼び出しを、関数の中から行なうためには、延々と関数を書き続けることになります。

```
void myfun() {
   int a = 5;
}
void call_myfun() {
   myfun();
}
```

今度は、call_myfun()を呼ぶ関数を作らなければ、作業は始まらない。

そこで、自動で始まる、という関数が 1 個用意されています。これは名前も型も決まっています。

これをメイン関数といい、これを作っておくと、プログラムは、メイン関数から自動的に始まります。

先ほどの、call_myfun関数の名前を、mainに名前を変更し、型も指定されているintに変更、

中身として、呼び出し文を書きますが、同時に、リターン文を置きます。

戻り値はイントにしなさいという指定があるので、先頭に、int　と書きます。

そして、実際に何を戻すかを明示します。

ここでは、全て首尾よくいきました、を示すものとして、ゼロを書き込みます。

```
void myfun() {
   int a = 5;
}
int main() {
   myfun();
   return 0;
}
```

グロサリー２　文、ステートメント、**statement**

文とステートメントは同じ意味、文は、statementの訳語になります。
Cのプログラムは、

1. 文の並びで構成される。
2. 最低でも、関数が一つ必要。一つの時は、メイン関数となる。
3. 関数の中には、ゼロ個以上の文が入る。
4. 文は、関数の外に置くこともできる。
5. 文は、しっぽにセミコロンがついている。

Cのプログラムは、文の並びだ、とくれば、
文の書き方を知ってしまえば、クール間違いなし、です。
Cのような高級言語には、プログラムを、普通の文章を書くようにしたい、
という方向性があって、そのようなものを、いくつか用意しています。

1. イフ文
2. ワイル文
3. フォー文

イフ文は、
もし、～であれば、これを実行する。
そうでなければ、こっちを実行する。
プログラムの流れはこれと全く同じですので、一発で覚え、すぐ使えるようになります。
よく使う文については、個別にグロサリーを用意してあります。

グロサリー３　宣言、デクラレーション、**declaration**

宣言とデクラレーションは同じ意味、宣言は　declaration　の訳語となります。
プログラムでは、データを置く場所を確保し、そのデータを操作します。
データを置く場所とくれば、それは、メインメモリー。
データを置く場所を確保するとき、わたしたちは、その場所に、適当な名前をつけます。
宣言というのは、米国人の好きな、多少ユーモアを込めた壮大な言い方で、
実情は、こちらがつけた名前をCにお知らせしておく、という作業になります。
このとき、どんなデータを入れるのに使うのか、というデータ型もお知らせしておきます。
具体的には、
int cool;
これで、小数点のつかない、プラスまたはマイナスの数字を入れる場所を
coolという名前で使うことにしますので、よろしくというお知らせの完了です。
Cでは、早速、nametableのような場所に名前とこれを収めるメモリーのアドレスを書き込んでおきます。
この時点でのわたしたちの認識は、
名前はお知らせしたけど、値はまだ入れてないので、実際には、この場所は、
まだメモリー上には作られていないはず、となります。確かめてみましょう。

```
#include <stdio.h>
int cool1;
int main() {
    int cool2, cool3;
    printf("cool1: %d\tcool2: %d\tcool3: %d\n", cool1, cool2, cool3);
    return 0;
}
```

これを、ターミナルから、gcc -o cool cool.c　とします。結果は、$ ./cool
cool1: 0　　cool2: 0　　cool3: 0
全てゼロで初期化されてちゃんと存在しています。
場所（変数）に値を入れることを代入（アサインメント）といいます。
gccでは、int cool; とあれば、それは、int cool=0;　として扱ってくれます。
ここでは、ローカル変数も同様にゼロで初期化されていますが、メイン関数でのふるまい、
ということで、メイン以外の関数では、いつものように、初期化をして下さい。

グロサリー４　代入、アサインメント、**assignment**

代入、アサインメントは、同じ意味、代入は、assignmentの訳語になります。
今、int a=5;　という文があるとき、
int a=5; の　=　は、アサインメントオペレーター、代入演算子といいます。
算数のイコールが、つりあってる、同じだ、という静的な意味を持つのに対し、
Cでは、左側に書いた場所に、右側にあるものをコピーせよ、
というダイナミックなアクション命令となります。
つまり、Cでは、=　を算数とは別の意味に使っています。
なんで勝手に別の意味に使うのじゃ、という議論は、当然ありました。
パスカルでは、:= を代入の意味に使い、=　には算数と同じ意味合いを持たせています。
算数は、わたしたちを計算機械に見立て、何かやらせようとします。
a=5を見たら、aと5はつり合っている、つまり同じだ。
この瞬間、aは5で確定している、と見ます。
コンピューターは、自動化された脳をもっていません。
この状況を具体的に作ってやります。
aという名前で、数字を１個入れられる領域をメモリーに確保し、
そこに５という数字を投げ込みます。
とりあえず、小数点つきの数字は使わないので、型情報として、intを与えておきます。
int a = 5;
これでCは、プログラム中にaという名前が使われていることを認知し、
そこには数字の5が入れてあることを知っています。
これをやっておけば、aと５は同じか、つまり、aの中身は5か、という質問をすると
意味ある反応が期待できるでしょう。
左辺と右辺がつり合ってる、左辺と右辺は同じ、の意味には、==　を使います。
イコールサインを別の意味に使うというのは、乱暴なのでは、と思うこともありました。
でも現在は、これでいいのだ、と思います。
世界中のプログラマーが、長期にわたり　:=　とタイプ２回打つより、
１回のほうが断然能率的で、しかも、これは代入ですよ、という説明は、
意外なので、１回聞けばまず忘れません。

グロサリー５　型、データ型、**type**、**data type**

型、データ型は、type、data typeの訳語になります。
型とは、獲得したメモリーに、どのようなデータを入れるか、
それは、何バイトなのかを規定したものです。
例えば、イント型は何バイトになっているかは、マシンにより、異なる場合があります。
アップルのスノーレパードでは、ポインター型は、８バイトとなっています。
型はいろいろありますが、いつも使う型は、以下のように決めておきます。

1. int　イント型、　４バイト、小数点のつかない数字はこれを使う。
2. char チャー型、　１バイト、文字１個。
3. float フロート型、４バイト、小数点付きの数字、クセでこれを使ってしまう。
4. double ダブル型、８バイト、小数点付きの数字、本来はこちらを使う。
5. * ポインター型、　４バイト、例えば、イントにこの星マークをつけると、
   イント型のポインターになる。int* myptr;
6. プログラマーが自分で作る型、通常はポインターなので、４バイト。

余談ですが、charは、characterを短縮したものなので、カー、もしくは、
ケャーのように発音する人もいます。日本のようにキャラという人も、
少数ですが、いないわけではありません。
ですから、お好きになさって下さい。

グロサリー６　式、エクスプレッション、**expression**

式とエキスプレッシヨンは同じ意味、式は、expressionの訳語です。
式のしっぽにはセミコロンがありません。
従って、文の構成要素として使うことになります。
Cでは、3+4 を、独立で置くことはできません。
わたしたちは、習性で、頭の片隅に、3+4とか、答えの７とかを置くことが出来ます。
Cには、頭というものがないので、メモリーに場所を確保、
3+4の結果をそこに入れるといった手続きが必要です。
例えば、ルビーのirbなら、その場で答えを出します。
$ irb
>> 3+4
=> 7
何度でも実行可能な高速バイナリープログラムを作る、という必要がなければ、
irbは便利です。
式は、算数の式と、とても似ています。
というのは、算数の式を書けるようにしたいという願望から、実装されたものだからです。
アセンブリー言語は、意外にも、３＋４のような書式をサポートしていません。
代わりにADDというコマンドを使いますが、同時に２つの即値は使えません。
つまり、ADD(3, 4)ではエラーになります。
そこで、MOV (3, EAX)　即値3をレジスタEAXにコピー。
ADD(4, EAX) EAXの中身3と即値４を加算、結果をEAXに置く、のように書き込みます。
初期のコンピュータの先生は数学科の出身の方も多く、
これでは、あんまりではないかということで、
より便利な言語開発の動機のひとつとなりました。
式のカテゴリーに入るものは、
1.単項のマイナス、−４など。＋５のプラスは、無視されます。つまり書かないのと同じ。
2.単なる数字、512など。
3.比較の記号が入ったもの、3<4、3>4、3<=4、3!=4 など。
4.計算の記号が入ったもの、3*4、2/4 など。
5.ビット演算の記号が入ったもの、0 & 1、0 | 1、1 ^ 1 など。
6.同じを示す記号の入ったもの、3==4

式を計算、吟味してみる。

式は、計算、吟味ができます。
a-5であれば、aに入っている値を探し、そこに入っている値が、8であれば、
3という結果を出します。
3<4であれば、これが成り立っているか、吟味します。具体的には引き算をします。
結果がマイナスならこれは成り立っているので、フラッグに 1 を入れておきます。
成り立っていなければ、フラッグにゼロが入ります。

Cが用意する、イフ文、ワイル文などでは、式を吟味スタイルで利用します。

```
if (3+4)
    puts("Hello");
```

この場合、イフの後ろのかっこの中の式を計算します。
結果は 7 ですが、この結果を 2 択のtrueかfalseに置き換えるために、吟味します。
falseとは、結果がゼロの場合であり、trueとは、それ以外のすべてのケースです。
ここでは、結果は 7 ですから、trueであり、ハローはプリントされます。
if (3-3) だと、結果はゼロなので、ハローはスキップされ、プログラムは次の行に進みます。

グロサリー７　ポインター、**pointer**

ポインターに訳語はないようです。ポインターのスペルはpointerです。
今、int i = 5;　とあれば、
イント型の数字を保持する場所１個を、メモリー内のしかるべき領域に確保、
その場所の名前は、iで、そこに即値５をコピーする、
という操作の指示になります。
次に、１個ではなく、例えば６個分連続して確保したいとします。
この場合の本来の方法は以下のようになります。
それには、ライブラリー関数、malloc()を利用して、メモリーの配分を受けとります。
メモリーに使う単位系は、バイトです。
イント型は、１個につき４バイト必要なので、４x６、２４バイトもらうようにします。
さて、ここで問題です。メモリーをどうやってもらうんでしょうか、
もらったメモリーをどこに保管するのでしょうか。
感じとしては、メモリーを保管するための別のメモリーが必要なのでは、と思います。
だとすると、堂々巡りが始まるのでしょうか。
メモリーを受け取るときの感じは、ツアーの幹事が代表して、利用するホテルの部屋
番号をうけとり、その時、先頭の部屋番号のみを知らせてもらえば、そこから連続しての
部屋割りになっているので、クールというわけです。
ホテルの部屋番号のことを、ここでは、メモリーアドレスといいます。
ポインターは、このメモリーアドレスを専門に保持する為のものとして、
用意されています。
ポインターには、適当な名前をつけることができるのは、いつもと同じですが、
最初に登場させる時には、名前の前に星印をつけます。
データ型は、配分を受けたメモリーを何に使うかを入れておきます。
ここでは、イント型の数字を入れるので、イント型とします。
実際の操作はあっけないもので、
int *p = malloc(24);
これで、イント型の数字６個用の場所を連続して確保、その先頭アドレスは
ポインター　ピーの中身として、しっかり持っていることになりました。

ポインターを操作して値を入れていく。

実際に値を入れていきます。

```
*(p+0) = 100;
*(p+1) = 200;
*(p+2) = 300;
*(p+3) = 400;
*(p+4) = 500;
*(p+5) = 600;
```

メモリーはバイトで、イントは４バイト、なので４つ飛びするのでは、
と考える方は天才的ですが、イントだよ、と宣言しておいたおかげで、
Cがよろしくやってくれます。
うまくデータが操作できたか、プリントしてみます。

```
int i;
for (i=0; i<6; i++)
	printf("%d\n", *(p+i));
```

ここで確認すべき問題が１つあります。
マロックしたときは、必ずフリーをしなければならない。
これをしないとメモリーリークがおこるかもしれない。
などと脅しをかけられるので、マロックは使いたくないと云う方が多いのです。
コンピューターは電源を落とせば、メモリーは残念ながらチャラになってしまいます。
従って、この脅しは、マシーンがワークステーション、サーバーのように
電源を落とさないことを前提にしているということです。
次にわたしたちのプログラムは、メイン関数ですぐに終了してしまいます。
なので、メモリーはチャラになります。
しかし、学校では、これはマナーですということで、フリーしなさい、といいます。
そこで、free(p); として下さい。
このとき、ポインターが保持している先の中身には用はなく、
ポインター自身を消去するので、星印はいりません。
ポインターは難しいと感じる方がとても多いので困ってしまいます。
日本語の訳語がないからでしょうか、
記憶領域内場所保持子では、どうでしょう、余計わからない、ごもっとも。

マロック関数を使ったプログラム全体では。

プログラム全体では、

```
// malloc.c
#include <stdio.h>
#include <stdlib.h>
int main() {
    // we can do malloc( sizeof(int) * 6 )
    int *p = malloc(24);
    *(p+0) = 100;
    *(p+1) = 200;
    *(p+2) = 300;
    *(p+3) = 400;
    *(p+4) = 500;
    *(p+5) = 600;
    int i;
    for (i=0; i<6; i++)
        printf("%d\n", *(p+i));

    free(p);
    return 0;
}
```

コンパイルは、アップルでは、-arch i386　を加えておきます。

```
$ gcc -arch i386 -o malloc malloc.c
```

ポインターについてまとめます。

1. ポインターは、変数のアドレスを保持するための、専用の変数。
2. 宣言のとき、宣言と初期化兼用のとき、関数本体の引数のとき、
   ポインターが指している変数の中身を意味したいとき、先頭に*をつける。
3. 典型的な使用例はマロック関数、これを何度か練習しておけば、ポインターをすっと使えるようになる。

グロサリー８　配列、アレイ、**array**

配列とアレイは同じ意味、配列はarrayの訳語になります。
アレイは、兵隊さんが整列している、同じものがズラっと並んでいる、
そんなイメージを持ちます。
今、数字１個分の領域を確保して、そこに100という数字を入れるなら、
int i = 100;　とします。
複数の領域を連続して確保する、ただしその入れ物に値を入れない場合は、
int arr[6];
複数の領域を連続して確保する時、値も同時に入れる場合は、
int arr[] = { 100, 200, 300 };
この時、名前の後の鍵括弧に数は入れません。
そして、値の最後尾にはカンマはつけません。

Cのデザイナー、デニス・リッチーによれば、
配列の書式は、ポインターの操作を使いやすくした、シンタックスシュガーということです。
Cの前には、Bがあり、BからCへの過渡期に、NB(New B)も存在しました。
NBの書式は、
int iarray[10];　これは現在と同じです。
int ipointer[];　現在では、星印に変更されています。
配列はCが１０個分のスペースを用意してくれるのに対し、
ポインターは、プログラマーが手動でスペースを確保するというのも現在と同じです。
配列データへのアクセスは、*(iarray+1)
そして、iarray[0]
というやりかたも、シンタックスシュガーとして、Bに登場しています。
実際に操作を行ってみます。
イント型の配列、nums[]を作り、これに、100, 200, 300　と入れておきます。
int nums[] = { 100, 200, 300 };
ポインターをそのまま書き込むと、
int *p = { 100, 200, 300 };
これはエラーになります。この場合は、グロサリー７にあるように、malloc()関数を使います。
ナムスにデータがうまく収まったか、プリントしてみます。

配列のデータをプリントする。

printf("%d\n", *nums);
printf("%d\n", *(nums+1));
printf("%d\n", *(nums+2));
シンタックスシュガースタイルでは、
printf("%d\n", nums[0]);
printf("%d\n", nums[1]);
printf("%d\n", nums[2]);
int nums[] = { 100, 200, 300 };　に対し、サイズを調べます。
int i = sizeof(nums);
printf("%d\n", i);　// 12 came out
１２になりました。イントの数字４バイトx３で、１２バイトです。
例えば、この配列を関数の呼び出しの時、引数として渡したい場合、
ポインターは、４バイトなので、ポインターで渡せば効率的です。
Cでは実際、自動的にそのように処理してくれます。

```
// ptr.c
#include <stdio.h>
#include <string.h>
passp(char *p) {
   printf("%s", p);
   strcpy(p, "No, you are not\n");
}
int main() {
   char name[] = "Hello, I'm Superman\n";
   passp(name);
   printf("%s", name);
   return 0;
}
```

1. メイン関数にて、配列nameを作った。
2. これをpassp()関数の引数として渡した。
3. この時、passp()関数はチャー型のポインターを受け取るようになっている。
4. 受け取ったポインターをプリント、ライブラリ関数、strcpyを使ってpの中身を書き換える。
5, pの中身を書き換える時、元の字数より多くしてはいけません。
6. passp()関数から戻ってきたら、配列nameをプリントしてみる。

グロサリー９　イフ文、**if statement**

イフ文は、if statementの訳語になります。
コンピューター言語の基本的コンセプトのひとつに、
一般的な英語の文章のように記述すれば、それがプログラムになるなら、
ただプログラムを眺めていれば、なにをするのか分る、
だったら、マニュアルも必要なし、これは便利だ、というのがあります。
if文の構成は、

```
if (式)
   文
```

さらに

```
if (式)
   文
else
   文
```

グロサリー６にも式の説明がありますが、式は吟味されます。
吟味とは、エバリュエーション（evaluation）の訳語です。
ここでは、式を調べ、この式は成立しているか、していないかを、
trueかfalseで知らせます。

```
if （~が成立しているならば）
   これを実行
else　それ以外では
   こちらを実行
```

このスタイルは、条件に応じて実行する文が異なるということから、
条件分岐、conditional branchと呼ばれます。
イフ文の例を示します。

```
int a = 7;                  // 変数aに即値７をコピーしておく
if (a == 7)                 // aは7と同じが成立しているか吟味する
   puts ("OK");             // 結果が真の場合はOKを表示
else                        // 結果が偽の場合
   puts ("Not seven");   // Not sevenと表示
```

グロサリー１０　ワイル文、　**while statement**

ワイルには、その間はずっと、といった意味があります。
while (式)　文
括弧の中に書かれた式を調べますが、それは結果として数字が取り出せる式
とか、不等式を置きます。
この式が成立していたら、本体部分をずっと実行して下さいという文になります。
最初にカウンターを作ります。
そして、ゼロから４まで、５回Coolをプリントさせます。
カウントを１ずつ増加させて、cnt < 5 という不等式が
成立しているか、毎回確認し、成立している間は、作業を
繰り返します。
1を加算するという時は、
cnt++;
という書式を使います。

```
int cnt;
while ( cnt < 5 ) {
   puts("Cool");
   cnt++;
}
```

cnt++の書き忘れは、無限ループになってしまうのであせります。
しかし、そこはあわてずに、コントロール+Cキーを押して下さい。

グロサリー１１　フォー文、**for statement**

コンピューター言語は、作った方のパーソナリティー、センスがとても反映されます。
ケン・トムソンがユニックスを立ち上げたマシン、PDP-7には、メモリーが24K積まれていた、
などと云います。
24KのメモリーでOSをあげてしまうとは、何なのでしょうか。
それは単なる神業として、そんな時、ケンはスペース・トラヴェルというゲームを作り、
PDP-7に入れて遊んでいました。
それは、ベル研究所での遠い昔の出来事として今に伝えられています。
Cの文法には、人間の言語ほどではありませんが、ちゃんと例外があったりします。
for　(式１;　式２;　式３)　文

for文をつくります。
1. カウンターを用意します。
int i;
for文の括弧の中は、３つに仕切られており、
その仕切り板としてセミコロンを使っています。
for (; ;)
　本体の文;
2. 括弧の中の最初の部分で、初期化をします。
for (i=0 ; ;)
3. 真ん中の部分に、調査すべき条件を置きます。
for (i=0 ; i < 5 ;)
4. 最後にカウンターの加算を置きます。
for (i=0; i < 5 ; i++)
5. 本体を書き込んで、完成です。
for (i=0; i < 5 ; i++)
　puts("Hello!");

グロサリー１２　プリントエフ、　**printf**

プリントエフ、プリントエフ関数、printf

```
int i = 100;
printf("%d\n", i);
```

これは、Cが用意するライブラリー関数プリントエフ()の呼び出しです。
この関数が、ある日、消えてしまったら、どうするの。
この関数のクローンを作るのは大変です。
これから、ものすごく利用することになりますので、
使い方を早いとこ覚えたいと思います。

ここで、重要な常識の確認をしておきます。
今、ここに、100 + 200　とタイプしました。
わたしたちは、これを見て、300　という答えを出すことに、不思議はありません。
しかし、わたしたちが、現に目にしているのは、活字とかフォントというようなものでして、
100は、フォントの１、フォントのゼロ、フォントのゼロと、プリントされているのは活字です。
活字を計算しても期待される結果は、出ません。つまり、これは文字なのです。
これを足し算した時、考えられる結果は、100200 です。
今、プログラムで、計算した値が300という時、
コンピューターの中で、計算などしていた数字を、
誰かが、文字の３、文字の０、文字の０のように変換してからプリントする必要があります。
つまりフォーマットを変更する必要があるのです。

プリントエフは、活字に変換してくれる。

プリントエフ関数は、プログラムが扱うデータを、
プリント可能な状態に変換して、ターミナルにプリントします。
プリントしたいものは、"　"の中に置きますが、
変換すべきデータは、%に続けて、そのデータ型に応じたオプションを書きます。
そして、データそのものは、コンマで区切り、"　"の後に並べます。
例:
printf("result: ");
result:　と表示される。

int i = 300;
printf("result: %d", i);　result: 300　と表示される。
dはデシマル、%iでも同じ結果となる。

int i = 300;
printf("result: %d\n", i);　result: 300　と表示され、そして、改行する。
\n　は、new lineの意味、改行される。
\は、エスケープシーケンスとなり、\とnはプリントされず、改行されます。

float fnum = 3.14;
フロート、ダブルは、%f　とする。
%3.2 とは、表示される数字は全部で３個、そのうち、小数点以下は２個表示すべしという指示。
printf("%3.2f\n", fnum);

char ch = 'A';
チャーは、%c
printf("%c\n", ch);

char str[] = "Hello";
文字列は、%s
printf("%s\n", str);

グロサリー１３　グローバル変数、大域変数、**global variable**

グローバル変数は、銀座４丁目にある和光の時計です。プログラムのどこからでも利用できます。
和光の時計と違うのは、どこからでも変更できるという点。
グローバル変数の作り方。
プログラムの先頭付近に、関数の外で宣言します。
関数の外で宣言した変数は、グローバル扱いになります。

```
int gvar;
```

グローバル変数を使う利点は、
1. 上を見れば、そこにあるので、関数の呼び出しで、引数として渡す必要なし。
2. どの関数からでも利用できる。
3. リターン文を書く必要なし。
4. 関数の中で変数の宣言など、必要なし。ただ使えばよろしい。
5. プログラムが終了するまで、変数は、そこに存在し続けるので、連続した作業に適する。

例を見ます。

```
// gvar.c
#include <stdio.h>
int gvar;
void print_gvar() {
    printf("gvar: %d\n", gvar);
}
void add_ten() {
    gvar += 10;
}
void add_hund() {
    gvar += 100;
}
int main() {
    print_gvar(); add_ten(); print_gvar(); add_hund(); print_gvar();
    return 0;
}
```

グロサリー１４　小数点の数字、**float**、**double**

小数点の数字を扱う操作は、本当は、かなり難しく感じられます。
ですから、ここは簡単に通過いたします。
小数点の数字を扱うデータ型は、３種類あります。
わたしたちは、習慣で、つい、floatを使いますが、本来は、doubleを使うべきかと思います。
インテルでは、floatで、データを作っても計算する前に、ダブルに格上げになります。
ですから、素直にダブル型を使うことにします。
ダブル型は、通常８バイト、つまり64ビットのサイズになります。
ご自分のプラットフォームでは、ダブルは何バイトなのか、１度調べておきましょう。

```
// float.c
#include <stdio.h>
int main() {
    printf("sizeof float: %d\n", (int)sizeof(float));
    printf("sizeof double: %d\n", (int)sizeof(double));
    printf("sizeof long double: %d\n", (int)sizeof(long double));
    return 0;
}
```

結果は、

```
sizeof float: 4
sizeof double: 8
sizeof long double: 16
```

となりました。
特に高精度の値が必要な場合以外は、ロングダブルは使わなくてもいいはずです。
信頼性の高い浮動小数点の計算を行うのは、プログラマーのウデがものをいいます。
当初は小数点以下３～４けたぐらいの、かわいい数字で、慣れてから先に進みましょう。

グロサリー１５、チャー型、**char**

チャー型、キャラクター、char、character
Cの文字の操作は、文字操作の原点です。
キャラクターというと、アニメのキャラクターを考えてしまいますが、
Cのキャラクターは、キーボ–ドから打ち込まれてくる１文字のことです。
データ型の名前は、characterを短くした、charを使います。

```
char ch = 'A';
```

１文字を表す時、クオートをつけます。
キーボードから１文字打ち込むと、それは、数字に変換されます。
どんな数字に変換されるかは、アスキーコード表がサイトにありますので、
これを見ると確認できます。
http://ja.wikipedia.org/wiki/ASCII
http://www9.plala.or.jp/sgwr-t/c_sub/ascii.html
キーボードから文字の入力を受け取り、10進数、16進数でプリントします。

```
// char.c
#include <stdio.h>
int main(int argc, char** argv) {
   int c;
   printf("Enter a character, quit for Ctrl_D\n");
   c = getchar();
   while ( c != EOF ) {
      printf("decimal: %d\n", c);
      printf("hex: %#x\n", c);
      c = getchar();
   }
   return 0;
}
```

Aをタイプ、リターンすると、Aの65とリターンの10がプリントされます。

グロサリー１６　文字列、ストリング、**string**

文字列とストリングは同じ意味、文字列はstringの訳語になります。
チャー型の配列を、文字列、ストリングと呼び、様々なサービスが用意されています。
このサービスを利用する為に、この１行を加えます。

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

1. 初期化の時点で、文字列を書く場合は、サイズを書きません。
   それでも書いたら、どうなるでしょうか。
   次の１行のサイズは、13文字です。
   "Hello!, world";　これを12しか確保しないでやってみます。

```
char s1[12] = "Hello!, world";
printf("%s\n", s1);
```

   この場合、エラーは出ませんが、
   プリントは、Hello!, worl　となります。そして、サイズの表示も12となります。
2. チャーの配列を宣言する。使う予定より少し多めに確保しておきます。

```
char s2[64];
```

   これにストリングコピー関数を利用して、"Hello!, " をコピーする。

```
strcpy(s2, "Hello!, ");
puts(s2);
```

   こんどは、ストリングキャット関数を利用して"Hello!, "のお尻に、"world"を追加する。

```
strcat(s2, "world");
```

   ここでのキャットは、ねこちゃんではなく、コンケイトネントの略で、連結するという意味。
3. チャーの配列に同じ文字列が入っているか調べる。
   これには、ストリングクンプ関数を使う。クンプは、コンペアを短くしたもの。
   この関数で注意するのは、同じと判定された場合、ゼロが返ってくること。

```
char s3[] = "Aloha";
if ( strcmp(s3, "Aloha") )
        puts("same");
else
        puts("not the same");
```

   この場合、not the same がプリントされる。なぜでしょうか。

ストリング・クンプ関数の戻り値は。

ストリングクンプ関数は、比較して同じだとゼロが返ってくる。
cmpは、compareを短くしたもの。
正確にいいたい方は、ストリングコンペアと発音して下さい。
if文の括弧の中が、ゼロとなると、それは、falseを意味するので、制御はエルスに行く。
それで、not the same がプリントされる。
もしも、ストリングクンプ関数の戻りがゼロならば、と修正する。

```
char s3[] = "Aloha";
if ( strcmp(s3, "Aloha")    == 0 )
        puts("same");
else
       puts("not the same");
```

グロサリー１７　ディファインとタイプデフ

ディファインとタイプデフ　define and typedef
タイプデフは、Cの型を自分の使い易い名前でも通用するように出来るサービスです。
チャー型は、１バイトのデータを保持できますが、マイナスの値にも対応しますので、
実際に保持できる値の範囲は、-128 … 127　となります。
いま、着メロのmidiデータを作ろうと思ったとき、
マイナスの値は、使わないので、符号なしチャー型を使おうとします。
符号なしチャー型なら、0 … 255 までの値を保持できます。
unsigned char mycha;
アンサインド　チャーは、名前が長いので、別の名前に変えたいと思います。
そこでタイプデフの登場です。
typedef unsigned char byte;
これをプログラムの先頭付近に書き込んでおきます。これで、
byte b = 128;
Cが利用するヘルパープログラム、Cプリプロセッサーにも、
同じような便利なサービスが用意されています。
Cプリプロセッサーのデファインを使うと、
#define unsigned char BYTE
BYTE b8 = 255;

グロサリー１８　構造体、**structure**

構造体型は、例えば、イント型２つと、チャーの配列を一つのパッケージとしてまとめ、
そのパッケージに名前をつけます。
組み合わせは、必要に応じてユーザーが決めます。
構造体は、ストラクチャーの訳で、キーワードは、これを短くした、struct　です。
書式は、

```
struct osoba_t {
   char name[32];
   float rate;
};
```

おそばを買ってきて、食べた後で、おいしさ度を入れてみます。
ここで注意すべきは、以下のことです。
1. これは、構造体型osoba_tの宣言であって、ひな型のようなもの。
実際のオブジェクトを生成するには、
a. 最後の}と;の間に、変数の名前を置く。

```
struct osoba_t {
   char name[32];
   float rate;
}soba1;
```

b. 別の行に改めて、変数を作成する。

```
struct osoba_t {
   char name[32];
   float rate;
};
struct osoba_t soba1;
```

a. のスタイルを使って、変数はsoba２以外作らなければ、osoba_tという構造体型の名前は、省略できる。

```
struct {
   char name[32];
   float rate;
}soba2;
```

構造体型のデータにアクセスするには。

構造体型のデータにアクセスするには、変数の名前の後にドットをつけて、
内部の名前を続ける。

```
strcpy(soba2.name, "yabu");
soba2.rate = 7.5;
```

構造体型を作成するとき、タイプデフも使うと、
変数を作る時に、ストラクトと書くのが省略できるので、すっきりします。

```
typedef struct {
    char name[32];
    float rate;
} osoba_t ;
```

これで、この構造体型の型名は、osoba_t型として、使えます。
osoba_t型の変数を作る。

```
osoba_t soba3;
```

作ったオブジェクトの変数に、値を入れる。

```
strcpy(soba3.name, "sarasina");
soba3.rate = 8.5;
```

グロサリー１９　ターナリー、**ternary**

謎めいた雰囲気はターナリー。
ターナリーとは、３つという意味。
Cでは、クエスチョンマークとコロンを使って、これを表現します。
1. クエスチョンマークの前に、調べるべき条件式を置く。
2. 結果がtrueなら、クエスチョンマークのすぐ後の値を選ぶ。
3. 結果がfalseなら、コロンの後の値を選ぶ。
条件式？値 : 値
condition ? value if true : value if false
例を見ます。

```
#include <stdbool.h>
#include <stdio.h>
int main() {
    bool boo = true;
    int n = boo ? 100 : 1024 ;
    printf("n: %d\n", n);
    return 0;
}
```

マクロでは、マックス、ミンは典型的な使い方です。

```
#define   MAX(a,b)  ( ((a)>(b) ) ? (a) : (b) )
#define   MIN(a,b)  ( ((a)<(b) ) ? (a) : (b) )
```

これをプログラムの先頭付近に書き込みます。
利用する時は、

```
int curmax = MAX(100, 200);
printf("current max: %d\n", curmax);
```

ターナリーは、暗号みたいな、楽しいサービス。
１度覚えたら意外と忘れないので、良い書式なのだと思います。
どんどん使って楽しみましょう。

グロサリー２０　変数、バリアブル、**variable**

変数とバリアブルは同じ意味、変数はvariableの訳語となります。
算数では、変数yは、数字のことです。
y=2x+4 のような場合、xもyも数字になります。
Cでは、変数は場所になります。
具体的には、実メモリーのどこか、適当な領域に名前を付けて管理します。
Cでは、変数がどこに作られるかについては、ハードウェアによるとして、
きっちり決めているわけではありません。
それでも、ストレージ・クラスというものがあって、
そこには、オートとスタティックという予約語が見てとれます。
しかしこれらは、通常、明示しなくてよいようにしてあるため、
プログラマーは、そんなものがあるのを気がつかなかったりします。

1. 変数が最初に登場する時、それが関数の中である場合、
   特別な明示をしないものは、すべてautoであるとみなす。
2. 変数が最初に登場する時、それが関数の外である場合、
   staticとなる。
3. autoは、関数の外で使うことはできない。
4. auto変数の寿命は、その変数が宣言されたところから、関数の終わりまで。
5. static変数は、プログラムが終了するまで、存在する。
6. 関数の中で宣言されたstatic変数は、同様にプログラムが終了するまで、存在する。
7. オートという意味は、プログラマーが手作業で、メモリーの配分要求、削除の操作などを
   やらなくていいということ。

今、関数myfun()を作り、そこに64バイトの配列を作り、値を入れました。
これは、ディフォルトでオートになるので、
この関数myfun()から抜ける時、この配列は消えることになります。
例えばこれは、メモリーの節約になっているのでしょうか。

ローカル変数は、メモリーの節約になっているのでしょうか。

OSが立ち上がって来た時、インテルのスタックには、８１９万２千バイト、
８メガバイト強のメモリーが配分されます。
先ほどの64バイトは、このスタックのトップからスタックポインターの位置を
動かし、使ったあとは、又スタックポインターをスライドさせて、
次の利用に備えます。
しかし、８１９万２千バイトのスタックは電源がオフになるまでそこに存在します。
Cを開発する時点での、ケンのマシンPDP-7に、メモリーは殆ど積まれていませんでしたので、
スタックの使い回し以外の方法は、ありえなかったのかもしれません。
スタックにはこれ以上メモリーを配分する必要がないので、
メモリーが少ないとき、気分的には楽です。
何はともあれ、スタックを使って処理するという方針は、
現在にそのまま受け継がれ、スピードも早く、みんな大好き、というのは、驚異的です。
パスカルのワース先生は、誰かさんの好きな、当時、世界第２位のスピードを誇る
シーモア・クレイが設計したスーパーコンピュータCDC6400を使っていました。
パスカルは、そんなこともあって、メモリーにかんして、もう少しおおらかな感じが
あります。
Cのスタック重視の考えは、現在では、特に有利ということではないと思いますが、
ローカル変数にするのは、大変すばらしいこと、というのは、伝統として、
完全に受け継がれています。

グロサリー２１　ジーリブをスラックスで使う。

このグロサリーは、Cを最初からジーリブライブラリー込みで
始める方に、Cの調子を見てもらうために用意したものです。
Cをジーリブ込みで使うには、ジーリブが必要です。
スラックスライブCDは、CentOSなどと同じように、
立ち上がれば、即ジーリブを使うことができます。
そこで、ここでは、スラックスライブCDを使って、
Cとジーリブを使う方法をご紹介します。
実際、USBペンドライブがあると全然便利ですが、ここでは、無しで行います。
ファイルのセーブするには、gmailの仕組みなどを設定しておく必要があります。
尚、詳細は、だってクールだ　SLAX をご参照下さい。

1. http://sourceforge.jp/projects/slax/downloads/44899/slax-ja-6.1.2-2.iso/
　　にて、アイソファイルを入手、CDに焼きます。
2. CDをトレイに置き、再起動、スラックスライブCDからブートします。
3. 立ち上がりましたら、KMenu -> KWrite で、プログラムを作ります。

最初のプログラムは、

```
// ghello.c
#include <glib.h>
int main() {
   g_print("Hello!, world\n");
   return 0;
}
```

コンソールを立ち上げ、

```
# gcc -o hello `pkg-config --cflags --libs glib-2.0` ghello.c
```

と入力。

```
# ./hello
```

グロサリー２２　ジーリブ用のメイクファイル

ジーリブを使うプログラムをコンパイルするときは、
gcc -o hello `pkg-config --cflags --libs glib-2.0` ghello.c
と長い１列になります。
プログラムを何回も修正する時、これでは不便です。
そこで、makeというプログラムを利用します。
メイクを利用するには、メイクが見に来るファイルを用意します。
ひながたを、適当なフォルダーに保管しておき、
使う時は、コピーしてから、以下の２カ所をタイプします。
PROGRAM= ここにこれから作るアプリの名前を書きます。
FILES= ここにコンパイルに使うCファイルの名前を書きます。
これをCファイルと同じ場所に置き、
make
とタイプ、リターンで、実行可能なファイルができあがります。
以下をコピー、Makefileの名前で保存して下さい。
先頭に#があると、その行は、コメントとなります。
$(CC) $(CFLAGS) $(LDADD) -o $(PROGRAM) $(FILES) のある行の先頭には、タブを入れて下さい。
rm -f $(PROGRAM)のある行の先頭には、同様にタブを入れて下さい。スペースではだめです。

```
# Makefile for gcc
PROGRAM=
FILES=
CC=gcc
CFLAGS=-g -Wall `pkg-config glib-2.0 --cflags`
LDADD= `pkg-config glib-2.0 --libs`
$(PROGRAM):$(FILES)
	$(CC) $(CFLAGS) $(LDADD) -o $(PROGRAM) $(FILES)

clean:
	rm -f $(PROGRAM)
.PHONY: clean
```

グロサリー２３　**C**プリプロセッサー　**C preprocessor**

Cプリプロセッサー、　C preprocessor
Cは、ヘルパープログラムを利用します。
このヘルパープログラムの名前が、Cプリプロセッサーとなります。
わたしたちは、Cプリプロセッサーを使う時、明示的に、指令を書き込みます。
この書式は、Cに対するそれとは、違います。
問題は、このCの書式とは関係ないものが、プログラムの先頭付近に、ドーンと置かれることでした。
#include <stdio.h>
これは、プリプロセッサーに、Cライブラリーのスタンダードアイオーに含まれる関数を、
これより使いたいので、必要な処理をして下さい、という指令になります。
プリプロセッサーが行う必要な処理とは、どんなことをするのでしょうか。それを見ます。
1. helloのような名前でフォルダーを作ります。
2. その中に、以下のような、hello.cプログラムを置きます。

```
// hello.c
#include <stdio.h>
int main() {
   puts("Hello, testing one two");
   return 0;
}
```

3. gccを以下のオプションをつけて呼びます。

```
gcc -save-temps -O3 -mdynamic-no-pic -S hello.c
```

これがうまくいくとフォルダーの中に、hello.s と　hello.i　が作成されます。
アイファイルは、プリプロセッサーが、スタンダードアイオーを展開し、
main()関数の前に置いてくれた状態を示しています。
これでメイン関数は、プットエスを利用できるというわけです。
アイファイルを見ていると、プリプロセッサーが、舞台裏で何をしているのか、垣間みることができます。
尚、hello.sは、アセンブリー言語に変換されたファイルとなります。

グロサリー２４　関数のプロトタイプ宣言。

関数のプロトタイプ宣言とフォワードデクラレーションは同じ意味。
スペルは、function prototype、forward declarationとなります。
関数は、本体と呼び出しの２本立てになっています。
プログラムを書く時の順番は、先に本体、後に呼び出しとします。
今、何らかの理由で、この順番が逆になってしまいました。
何らかの理由とは、ただそうしてみたい、でもよいのです。
問題は、これでCが警告を出すことです。
実際に調べてみます。

```
// proto.c
#include <stdio.h>
int main() {
    myfun();
    return 0;
}
void myfun() {
    int i = 5;
    printf("i: %d\n", i);
}
```

これを、コンパイルします。

```
$ gcc -Wall -o proto proto.c
proto.c: In function 'main':
proto.c:4: warning: implicit declaration of function 'myfun'
```

実行ファイルは作成され、動作もしますが、警告がある場合は、
原則、警告すべてに立ち向かい、きれいにします。
ここでの警告は、関数マイファン宣言が見当たりません、です。
そこで、関数マイファン宣言をします。具体的には、

1. 関数本体の１行目、波かっこの前までをコピー
2. プログラムの先頭付近にペースト。
3. 宣言文にするため、しっぽにセミコロンをつける。

プロトタイプ宣言を加えたファイルは。

```
// proto.c
#include <stdio.h>
void myfun();

int main() {
    myfun();
    return 0;
}
void myfun() {
    int i = 5;
    printf("i: %d\n", i);
}
```

これで、コンパイルします。

```
$ gcc -Wall -o proto proto.c
```

警告は消えました。