[プログラミング序論 II　4 回目　２００２・１０・3]

いろいろな変数：局所変数、大域変数、静的変数の使い方

関数の中で宣言されている**局所変数**（local variable）は、関数の中で一時的に使うデータを格納しておくのに使います。それに対し、関数の外で宣言された**大域変数**(global variable)はプログラム全体で使うデータを格納しておくために使います。

関数は数学の関数のように、パラメータのみを使って、値を計算するだけではありません。通常、関数はある機能ごとに作り、プログラムをわかりやすく部分部分に分けて書くために使います。大きなプログラムを作る場合にはこのようにして部分に分けてかくことは非常に重要です。例えば、カウンタがあり、それを増やしたり（increment）減らしたり(decrement)という関数を次のようにしてつくっておきます。

```
int counter;
void increment() { counter += 1; }
void decrement() { counter -= 1; }
```

これは非常に簡単な例ですが、このようにしておいて、この関数を使っておけば、counter に対する演算をいろいろなところにかくよりも、関数の名前、つまりやりたいことの名前でかくことができるようになり、プログラムがわかりやすくなります。この関数は値をかえしませんが (void をつかった手続き)、手続きに名前をつけて使うことができるようになります。

多くの局所変数は、大域変数でもよい場合があります。一時的に大域変数をつかってもよい場合です。例えば、int t；と大域変数を宣言しておけば、

```
int foo(x,y) {  t = x + y; return t; }
```

でもかまいません。しかし、大域変数は他の関数でも使われる可能性があるので、

```
int goo(x,y) { t = x-y; z = foo(x,y); …. ; return t; }
```

という場合には、関数 goo でセットした t の値が関数 foo で壊される（これが**副作用**!）ので、思いがけないエラーがおきてしまうことになります。これが大きなプログラムになると間違いを見つけるのが大変になってしまいます。局所的しか使わない一時的なデータにはできるだけ局所変数を使い、必要なときだけ大域変数を使うようにしましょう。

関数が終わっても値を保持したい場合につかう**静的変数**は他の関数には使われたくないが、値を保持したい場合につかいます。

引数の値渡し

下のプログラムを考えてみましょう。

```
main() { int i; i = 10; j = foo(i); printf("i=%d¥n",i); }
int foo(int i){ i = 100; return i; }
```

foo の中で、i に 100 を代入して、i の値を変更していますが、main の変数 i は変わりません。したがって、printf の結果は 10 となります。これは、main の i の値（つまり 10）が、関数 foo にわたり、関数のパラメータ変数 i にセットされます。パラメータ変数は関数の中では局所変数と同じに扱われるので、これを変更しても、逆に main の変数 i はかわりません。このことを引数の**値渡し**といいます。（教科書 178 ページ参照）

なお、配列や文字列の場合は、呼び出し側の配列が変わってしまいます。これについてはポインターのところで説明します。

> プログラミング言語によっては、値でなく呼び出し側の i 自体が引数となることがあり、この場合は i は 100 になってしまいます（例えば、FORTRAN）。これを**参照渡し、名前渡し**いう。言語によってことなる。

関数の再帰呼び出し

ある関数が、自分自身を呼び出すことを再帰呼び出し(recursive call)といいます(教科書 197 ページ)。例えば、階乗を求める関数は次のように書くことができます。

```
int factorial(int n){
    if(n == 0) return 1;
    else return n*factorial(n-1);
}
```

つまり、n!は、(n-1)!に n をかけたものというわけです。但し、これをどこまでも続けても止まらなくなってしまうので、１のときには１！＝１としています。階乗の定義は、１から n まで掛けたものということあれば、このようなことをしなくても、単にループで１から n までかければいいのであまり意味がわからないかもしれません。しかし、再帰呼び出しの考え方はもっと重要な意味をもっています。この

考え方を身につけることによって、いろいろな問題が簡単に解ける強力な考え方です。
では、有名なハノイの塔の問題を考えることにしましょう。ハノイの塔とは、３本の柱があり、一番左の柱に下から大きな盤、その上に順に小さな盤が乗っているというものです。問題は

１、一度に１枚の盤しか移すことができない（片手しか使えない）
２、小さな盤の上に大きな盤を乗せてはいけない（壊れてしまう）

この条件で、左の柱から、右の柱に移すというパズルです。中間的な置き場として真ん中の柱を使うことができますが、上の条件は満たしていなくてはなりません。
この問題の解き方としてまず、大小２枚の場合を考えてみましょう。これだったら、まず小さい盤を真ん中に移して、大きな盤を左に移し、最後に小さい盤を左に移せばよいことになります。では３枚ではどうでしょうか。この場合、まず、２枚の手順を使って、上２枚を真ん中に移します。それで、一番下にある盤を右の柱に移し、また２枚の手順を使って、真ん中から右に移せばいいということなります。

１、これを一般化して、n 枚の場合には、
２、n-1 枚をあいているところに移す。
３、一番下の番を目的のところ（右の柱）に移す。
４、n-1 枚を目的のところに移す。

という手順でやればいいということになります。これの解答に関しては教科書をみてください。n 枚の盤を移すという関数を定義して、その関数を再帰呼び出しを使うことで、エレガントにプログラミングすることができます。
もう、一つの問題を考えてみましょう。任意の整数を一文字の出力ルーチン(putchar)を使って、プリントアウトするという問題です。例えば、123 という数字の場合は”1”,”2”,”3”と出力するという問題です。一桁の場合であれば、単にその数字を’0’に加えることによって、出力します。では２桁はどうでしょう。この場合はまず、10 で割って、その商が２桁目なので、その数字を印刷します。で、あまりを１桁にしてプリントすればいいわけです。これを一般化して n 桁の数とすると、数 d に対し、

１、まず、一桁、つまり 10 よりも小さければ、putchar(d+’0’)
２、最下位以外の n-1 桁について、プリントする。つまり、d/10 を印刷。
３、最下位の桁、d%10 を印刷。

とすればいいわけです。

```
void print_number(int d)
{
  if(d >=10) print_number(d/10); /*  上の桁を出力  */
  putchar('0'+d%10); /* 最下位の桁を出力  */
}
```

再帰の考え方は、ある n について問題を解くときに、n-1 についての解答でとくことができるときに使うことができます。プログラムを考えるときに、特定のケース、特定の数についてプログラムを考えるだけでなく、それを一般的な入力、一般的な n について考えることはプログラミングについての非常に重要な考え方です。

小テスト問題

１、　２次元座標上の２点(x1,y1) (x2,y2)を表す数（不動小数点数）x1,y1,x2, y2 をパラメータとして、２点間の距離を返す関数 distance を定義しなさい。例えば、座標(1.2,2.0) (-1.1,3.4)の距離を求めたい場合には,右のように呼び出すものとする。なお、不動小数点数は double とする。平方根を求める関数は sqrt である。

```
main() {
  double d;
  d=distance(1.2,2.0, -1.1,3.4);
  printf("distance=%g¥n",d):
}
```

２、　上記と同じパラメータで、パラメータの２点を対角線とし、上下の辺が x 軸、左右の辺が y 軸に並行な四角形の面積を求める関数 area を定義しなさい。

```
main() {
  double a;
  a=area (1.2,2.0, -1.1,3.4);
  printf("area=%g¥n",a):
}
```

次回は、ポインタについて