[プログラミング序論 II　8回目　２００２・１０・３１]

構造体の代入、引数渡し、返り値、sizeof 演算子

構造体とは、いくつかのデータをひとまとまりにしたものです。プログラマにとってはコンピュータの中で何かをさせたい場合にそれを表現するためには必要な機能です。

前回の最後にデータ型について、説明しましたが、同じデータ型同士であれば、代入ができます。例えば、座標を表す point という名前の構造体定義されているものとしましょう。

```
struct point { int x,y; }; /* 構造体の定義 */
struct point A,B;
```

と宣言されていれば、以下のように代入できます。（演算はできません）

```
A = B;
```

代入とは、構造体のコピーです。

当然、引数にも使うことができます。関数 foo が以下のように定義されていれば、

```
void foo(struct point a, struct point b){ … }
```

これで、

```
foo(A,B);
```

と書けば、呼び出しに構造体をつかうことができます。以前、関数の説明をしましたが、C言語の場合は**「値渡し」**です。つまり、引数の値はコピーされて、引数として関数に渡されます。

また、関数の返り値としても返すことができます。例えば、以下のようにすることができます。

```
struct point goo(…) {  struct point X; …. retrun X; }
```

受け取るほうは、

```
A = goo(…);
```

として、返り値をコピーすることができます。構造体は、データ型ですので、int や double と同じようにつかうことができます。struct point と書くのが面倒だったら、前回、説明したとおり、typedef をつかって、データ型を定義しておけばすこしは書く量が減って、わかりやすくすることもできます。

```
typedef struct pont point_type;
point_type A,B;
```

とかけます。

以上、代入、関数の引数、返り値、どれも、構造体の場合も int や double と同じように**「コピー」**されます。コピーなので、大きな構造体の場合は時間がかかってしまうことになります。では、どのくらいのサイズなのか。データ型のサイズは、**sizeof 演算子**を使って調べることができます。例えば、int は、4 バイトなので、sizeof(int)は 4 になります。例えば、上の 2 つの整数をメンバーとしてもつ構造体 point のサイズをしらべてみましょう。

```
printf("size is %d¥n",sizeof(struct point));
```

を実行してみると、4 バイトの int が２つなので、8 とプリントされるはずです。

構造体とポインタ、malloc 関数

構造体を上のように、直接、引数に使ってしまうと、コピーされるため、不効率になってしまうことがあります。前の例では整数が２つだけなので、たいしたことはありませんが、100 バイトもある構造体の場合はいちいちコピーしていたら、遅いプログラムになってしまいます。（実際、このような小僧体の代入や引数は最初のC言語にはない機能でした。多くのプログラムでは構造体に引数を直接かくことは注意する必要があります）

このような場合は、ポインタを使います。ポインタはプログラムを効率的なプログラムを書くための機能です。ポインタの宣言は、**データ型のあとに＊**をつけたものですから、前の例の point 構造体へのポインタの宣言は、struct point *　で宣言します。前の例の関数宣言を書き直してみましょう。

```
void foo(struct point *ap, struct point *bp){ … }
```

呼び出しのほうはポインタを渡すわけですから、**&演算子**を使って、ポインタをつくって渡します。

```
foo(&A,&B);
```

ポインタからメンバーの値を参照するには、**->演算子**をつかいます。たとえば、ap で指されている構造体から、メンバーx を参照するには、

```
t = ap->x + 1;
```

と書きます。これは左側にかけば、メンバーへの代入になります。

```
ap->x = 123;
```

これは、まず構造体を参照して、そのメンバーを参照すればいいので、ap->x は、
```
(*ap).x
```
ともかくことができます。しかし、->のほうがわかりやすいので、こちらを使うようにしましょう。

以前の資料に書いておきましたが、メモリを確保する関数に malloc という関数があります。この関数は、malloc(確保するバイト数)で呼び出します。malloc 関数から返る値は、確保されたメモリのアドレス、つまりポインターです。使っているマシンの環境にもよりますが、malloc 関数は文字へのポインター、つまり、char *で、<stdlib.h>で宣言されていることがおおいので、必要なメモリを取ったあとは適当なデータ型のポインタに変えてやる必要があります。これをやるのが、**キャスト演算子**です。
```
ap = (struct point *)malloc(sizeof(struct point));
```
必要なバイト数は sizeof 演算子を使って計算しています。
このように、プログラムの中で実行中にメモリを確保するようなプログラムでは構造体へのポインタは不可欠な機能です。

共用体(union)

構造体はいくつかのデータをまとめたものでした。これに似た機能として**共用体**というものがあります。
```
union 共用体の名前　{
    フィールドのデータ型　フィールド名；
    フィールドのデータ型　フィールド名；
     …
};
```
struct が union に変わっただけですが、共用体はこのフィールドのデータのうち、どれかだけを持つものです。つまり、いくつかのフィールドでこのデータ型を「共用」するわけです。例えば、以下のような例を考えてみましょう。
```
union int_or_double {  int i;  double d;  };
union int_or_double x;
```
この x に対して、x.i = 100; として、i に 100 を代入します。この値は、x.i で参照することができますが、x.d = 12.13; とすると、この共用体は共用されていますので、x.i の値は上書きされて、使えなくなってしまいます。この共用体とは、実際にはフィールドのデータの中で、一番大きいデータのサイズをとって、そこを共用するためのものです。なので、メモリを節約したい場合につかいます。
最後に、struct と組み合わせた例を紹介します。
```
struct data {  int type;  /* type が、0 だったら、i を格納、1 だったら、d を格納　*/
  union_or_double i_d;  } t;
```
これは、type の値によって、union にはいっている内容を切り替える例です。たとえば、
```
if(t.type == 0) printf("int value is %d¥n",t.i_d.i);
else printf("double value is %g¥n",t.i_d.d);
```
とすることができます。

小テスト問題

1、　前回の成績を格納する構造体 sekiseki の配列
```
structure seiseki seiseki_kyou[50];
```
に５０人の成績が格納されているものとして、同じデータ型の構造体 average に、教科の平均値を計算し格納する手続き calc_average を書きなさい。宣言は、
```
void calc_average() { … }
```
となる。seiseki_hyou, average は大域変数として、すでに宣言されているものとする。

2、　前回の住所録を管理する構造体 personal_record の配列
```
structure personal_record records[100];
```
に 100 人の住所録が格納されているものとして、名前の先頭の文字が c で、年齢が k 以上の人数を返す関数 match を書きなさい。宣言は
```
int match(char c, int k){ … }
```
となる。なお、配列 records は大域変数として、宣言されているものとする。

次回は、最終回、入出力、プリプロセッサその他