構造体処理汎用モジュールの作成

恐神正博* ・ 西田富士夫**

Construction of Generic modules of structs

Masahiro Osogami ・ Fujio Nishida

Abstract

Recently, program modules reuseability has become necessary. A reuseable module must be easy to retrieve, expand and generally applicable to most case.

Struct is a very useful type of data for office computing operations, such as updating or retrieving data. But we need each module to perform the same processes despite of their structure differences. It would be very convenient to use generic modules library for automatic generation of modules as it requires only the own data structures.

In this paper, we propose a method of automatic generation of C function modules or C++ class modules from given data structures using the generic PROLOG module library.

1．まえがき

プログラムやモジュールの再利用について、近年盛んに研究や開発が行われている。再利用するためには検索しやすいこと、機能の追加や変更がしやすいこと、汎用性が高いことなどの条件が必要である。ここでは主として汎用性のある構造体利用モジュールの作成とその利用について述べる。

構造体は、登録、更新、検索などいろいろな事務処理などに広く用いられるデータ構造である。しかし、そのデータ処理のプログラムは、処理の内容が同じであっても構造体の構造の違いから再利用することができず、構造体ごとに作成されているのが実状である。もし、データ構造だけを与えれば、既成の汎用モジュールから標準的な処理を行うモジュールを自動的に作成できれば有用である。

この論文では、そのような汎用モジュールをプロログで作成しておき、これから与えら

*事務局　**経営工学科

れた構造体のデータ構造を用い、Cの関数モジュールや、C＋＋のクラスモジュールを生成する一つの方法について述べる。

2．C言語における構造体モジュールの再利用について

事務処理システムにおいては、その処理のほとんどがあるキーに基づいたレコードに対する項目の参照、更新またはレコードの追加といった処理であり、このようなプログラムの自動合成を考えた際、構造体の利用は必須の条件である。また、事務処理に限らずC言語のプログラム作成において構造体の利用頻度は非常に高い。

ところが、レコードの参照、更新といったような全く同様の処理を行う場合であっても、構造体の構造の違いにより、それらについて別々にモジュールを作成する必要がある。

レコードA

商品名 X(20)	価　格 9(7)	個　数 9(3)

```
void  readfile_struct_array(char *fn, struct item iteml[], int n){
      FILE *fp;
      int  i;
      fp=fopen(fn,"r");
      for(i=0;i<n;i+=1){
          fprint(fp,"%s", iteml[i].name);
          fprint(fp,"%d", &(iteml[i].price));
          fprint(fp,"%d", &(iteml[i].number));
      }
}
```

図1．レコードAの構造およびその一覧表示用構造体モジュールの例

レコードB

氏　名 X(20)	会員番号 9(5)	年齢 9(2)	電　話 X(20)	住　所 X(30)

```
void  readfile_struct_array(char *fn, struct meibo meibol[], int n){
      FILE *fp;
      int  i;
      fp=fopen(fn,"r");
      for(i=0;i<n;i+=1){
          fprint(fp,"%s", meibol[i].name);
          fprint(fp,"%d", &(meibol[i].code));
          fprint(fp,"%d", &(meibol[i].age));
          fprint(fp,"%s", meibol[i].tel);
          fprint(fp,"%s", meibol[i].address);
      }
}
```

図2．レコードBの構造およびその一覧表示用構造体モジュールの例

図１，２はどちらもレコードの構造とそのレコードの一覧を表示するための構造体モジュールの例である。ここで構造体処理モジュールread_struct_arrayには、データの入っているファイルを表すファイルポインタ、構造体の構造を表すポインタ、データ数という３つの引数を与えて処理を行うのは同じであり、呼出側の関数からみた場合、この２つは全く同様に扱うことができる。通常このような場合、生産性向上のため同一の関数を用いるが、構造体モジュールの場合、図１、２を見ても分かるとおり、メンバの数、タイプ等により若干の違いがあるため、どうしても構造体毎にモジュールを用意しなければならない。

データの構造と処理を条件として与え、プロログ上であらかじめ作成されたモジュールから、Ｃの構造体モジュールを自動作成するため、データ構造等の表現方法、またそれらの変換方法について順次説明を行う。

３．構造体とリストによる表現

構造体はＣやＣ＋＋のソースコードでは通常(1)のようにタグ名という構造体のタイプ名をTAGに続いてそのメンバーのタイプと名前{TYPEi NAMEi, i=1,...,n}を列挙して定義する。

```
struct TAG { TYPE1 NAME1; ...;TYPEn NAMEn;};          (1)
```

構造体のメンバに別の構造体を持たせることもできる。このように構造体のデータ構造は階層化することができ、それらの大きさも不定である。

一方プロログにおいては、データの種類としてリストを扱うことができる。リストにおけるデータ表現は、データの数、型、長さ等、すべて同一のリスト上に混在させることができ、また、リストの要素としてリストを持たせるなど階層的な構造を扱うことも容易であり、様々なデータ構造に対応させることが可能である。そこで、構造体のデータ構造をリストを用いて表現することを考える。

```
struct_member( LN, [L1,...,Lk]).                      (2)
```

ここでLNはリスト[L1,...,Lk]のリスト名で、構造体のタグ名を表す。Liはリストの要素で、メンバの名前、型、長さ等が表され、また、階層が下の構造体を表したものでもよい。例として図１，２で用いた構造体を、ＭＡＰＰ上で扱うことができるよう、式(2)に従ってリスト形式で定義してみる。

```
obj([mame(item1),type(struct(item))]).
struct_member(item,[ [[name,[20]],string],[price,int],[number,int] ]).
```

図３．図１のレコードＡの構造をリスト形式に置き換えたもの

```
obj([mame(meibo1),type(struct(meibo))]).
struct_member(meibo,[ [[name,[20]],string],[code,int],[age,int],
                      [[tel,[20]],string],[address,[30]],string] ]).
```

図４．図２のレコードＢの構造をリスト形式に置き換えたもの

このように、一般的な構造体についてリスト形式で表現することができる。先にも述べたように事務処理の多くは決められた処理以外行うことが少ない。次には、このようにリストによって表現されたデータ構造から、構造体データの読み込み、ソート等の一般的な処理について、プロログによりモジュールを作成する方法について順次説明を行う。

３．１　構造体データの読み込みの場合

図３，４のようなデータ構造の定義をあらかじめ用意しておき、以下の述語を用いていくことで構造体モジュールの作成を行う。

```
c(read_struct(X)):-obj([name(X),type(struct(TAG))]),
        struct_member(TAG,L),
        c(read_memberlist(pt,L)).                                    (3)
c(read_memberlist(X,[[HN,HT]|T])):-
        c(promptread__member(X,[HN],HT)),c(read_memberlist(pt,L)).   (4)
c(read_memberlist(X,[]).                                             (5)
c(prompt_read_member([X,MEM],TYPE)):-
        write_sp,writelist(['printf("',X,'->',MEM,'?= in',TYPE,'");']),nl,
        ptype(TYPE,PTYPE),write_sp,writelist(['scanf("',PTYPE,'",']),
        write_type_struct_obj(TYPE,[X,MEM]),write(');'),nl.          (6)
```

ここで、(3)の述語c(read_struct(X))のXには構造体名Xを指定し、図３，４のようにobj述語において定義してあるstruct_member述語との単一化により、タグ名TAGに対するメンバを述語c(read_memberlist(pt,L))のリストLに取り込む。(4)(5)はリスト要素の名前HNとタイプ名HT毎に再帰的に繰り返して処理することを示す。(6)は構造体の要素である各リスト要素毎に表示する入力プロンプト文と入力文がどのようにして作成されるかを示しており、ここにおいて対象言語の具体的なコードが記述される。ここにptype(TYPE,PTYPE)はint,floatなどのタイプと、%d,%fなどの入出力タイプ記号との間の変換述語であり、データ構造のデータタイプに対応させた形で展開させている。

以下に図３，４のデータ構造の定義を与えた場合得ることのできるモジュールをそれぞれ示す。

```
void   read_struct_array(){
   for(item *pt=i_ptr;pt<i_ptr+n; pt++){
      printf("pt->name=in string ");
      sacnf("%s",pt->name);
      printf("pt->price=in int ");
      scanf("%d",pt->price);
      printf("pt->number=in int ");
      scanf("%d",pt->number);
   }
}
```

図５．作成される図１のレコードＡの構造体データ読み込みモジュール

```
void   read_struct_array(){
   for(meibo *pt=i_ptr;pt<i_ptr+n; pt++){
      printf("pt->name=in string ");
      scanf("%s",pt->name);
      printf("pt->code=in int ");
      scanf("%d",pt->code);
      printf("pt->age=in int ");
      scanf("%d",pt->age);
      printf("pt->tel=in string ");
      scanf("%s",pt->tel);
      printf("pt->address=in string ");
      scanf("%s",pt->address);
   }
}
```

図６．作成される図２のレコードＢの構造体データ読み込みモジュール

３．２　ソートの場合

次にソートのついての説明を行う。ソートの場合においても、データ構造の定義は３．１で扱った図３，４のような形と同じである。すなわち、データ構造の定義は同じ構造体データに対しては重複して行う必要はなく、行いたい処理の提示のみでよい。

```
c(sort_struct(X,L,KEY_N)):-
 (member([[KEY_N,[S]],string],L),
  c(if(conv(greater(t(strcmp([X,KEY_N],[nxt(X),KEY_N])),t(0))),
  swaplist(X,nxt(X),L)));
  c(if(conv(greater(t([X,KEY_N]),t([nxt(X),KEY_N]))),
  swaplist(X,nxt(X),L))) ).                                    (7)
c(swaplist(X,nxt(X),[[LNH,LTH]|LT])):-
  swap(X,nxt(X),LNH,LTH),c(swaplist(X,nxt(X),LT)).             (8)
c(swaplist(X,nxt(X),[])).                                      (9)
swap(X,nxt(X),[LNH,[LSH]],string):-
  write_sp,writelist(['strcpy(tmp,pt->',LNH,');']),nl,
  write_sp,writelist(['strcpy(pt->',LNH,',',' (pt+1)->',LNH,');']),nl,
  write_sp,writelist(['strcpy((pt+1)->',LNH,', tmp',');']),nl.  (10)
swap(X,nxt(X),LNH,int):-
  write_sp,writelist(['tmp_num = pt->',LNH,';']),nl,
  write_sp,writelist(['pt->',LNH,' = (pt+1)->',LNH,';']),nl,
  write_sp,writelist(['(pt+1)->',LNH,' = tmp_num',';']),nl.    (11)
```

```
t([nxt(X),KEY]):-writelist(['(pt+1)->',KEY]).        (12)
t([X,KEY]):-writelist(['pt->',KEY]).                 (13)
```

先ほどと同様、(7)のsort_struct(X,L,KEY_N)の述語には(7)に先だって構造体名Xを与え、そのメンバをLに取り込む。(7)において配列要素の構造体と次の要素の構造体のソートのキーメンバについて比較するのであるがそのキーメンバのタイプが文字列か数値であるかにより、CやC＋＋では比較の関数が異なるのでこれに対応して場合分けしている。(8)(9)は各要素毎にswapを行う関数を作成する。(10)(11)はswapにおいて文字列を代入する場合と数値を代入する場合とで関数が異なりこれに対応して処理することを示す。

以下にそれぞれ図３，４のデータ構造の定義を与えた場合に得られるモジュールをそれぞれ示す。

```
void   sort_struct_array(){
   char      tmp[20];
   int       m,tmp_num;
   for(m=n-1;m>=1;--m){
      for(item *pt=i_ptr;pt<i_ptr+m; pt++){
         if(pt->number>(pt+1)->number){
            strcpy(tmp,pt->name);
            strcpy(pt->name,(pt+1)->name);
            strcpy((pt+1)->name, tmp);
            tmp_num = pt->price;
            pt->price = (pt+1)->price;
            (pt+1)->price = tmp_num;
            tmp_num = pt->number;
            pt->number = (pt+1)->number;
            (pt+1)->number = tmp_num;
         }
      }
   }
}
```

図７．作成される図１のレコードＡの構造体ソート用モジュール

```
void   sort_struct_array(){
   char      tmp[20];
   int       m,tmp_num;
   for(m=n-1;m>=1;--m){
      for(meibo *pt=i_ptr;pt<i_ptr+m; pt++){
        if(pt->number>(pt+1)->number){
          strcpy(tmp,pt->name);
          strcpy(pt->name,(pt+1)->name);
          strcpy((pt+1)->name, tmp);
          tmp_num = pt->code;
          pt->code = (pt+1)->code;
          (pt+1)->code = tmp_num;
          tmp_num = pt->age;
          pt->age = (pt+1)->age;
          (pt+1)->age = tmp_num;
          strcpy(tmp,pt->tel);
          strcpy(pt->tel,(pt+1)->tel);
          strcpy((pt+1)->tel, tmp);
          strcpy(tmp,pt->address);
          strcpy(pt->address,(pt+1)->address);
          strcpy((pt+1)->address, tmp);
        }
      }
   }
}
```

図８．作成される図２のレコードＢの構造体ソート用モジュール

４．構造体処理関数見出しの表示

事務処理における処理内容には、比較的画一的なものが多いことは前に述べたとおりであるが、データ構造の定義の他に当然処理内容についても指示を行わなければならない。

そこで構造体処理の関数について、どのような処理を行うことができるか一目で把握することができるよう、画面にメッセージ表示を行い、従来のＭＡＰＰの操作と統一するよう、日本語文で表示した呼出文表の述語を設ける。

```
struct_table([
p(1,['FILE,A,N'],['大きさ,N,の構造体配列,A,に,FILE,からデータを読み込む']),
p(11,['FILE,A,N,COND'],['ﾚｺｰﾄﾞ数,N,のﾌｧｲﾙ,FILE,を読み込み条件,COND,
        を満たす構造体,A,を表示し構造体配列retrに出力する']),
p(21,['A,N,MEM'],['大きさ,N,の構造体配列,A,のメンバ,MEM,の和を返す']),
･･･ ]).                                                    (14)
```

ここにp(NUM,VAR,CALL_STATEMENT)の３個の仮引数のうち、NUMは呼出文番号、VARはカスタマイズ用変数、CALL_STATEMENTは処理関数の呼出文である。

ユーザは所望の呼出文を番号で指定し、変数をカスタマイズした記号列STXをsp述語の引数部に記憶し、述語sp(SP)を作り、設計文書に取り込む。

設計文書の呼出文は以後のコード生成処理を効率的に行うため、(15)のように各呼出文毎に設けた、変換辞書述語conv_dirにより述語形に変換する。conv_dirの引数部は３個の項からなる。jp項は呼出分の日本語風表現と対応する述語表現を示す。proto項はＣの関数宣言やＣ＋＋のクラスにおけるプロトタイプ宣言を示す。funct項はメイン関数など、関数の手続き部における指定の手続きを表す。ところで、呼出文やその述語形において設けた引数は、ターゲット言語のＣやＣ＋＋の文法上の制限によりそれらの関数の引数部には全て取り込むことは困難なことがある。例えば検索条件の式や文字列である。ＭＡＰＰではこのような場合、呼出文取り込みの時、呼出文に含まれる変数はすべてカスタマイズして、関数の手続き部に取り込むが、ＣやＣ＋＋の関数で指定できるのはproto項で指定した変数だけである。

```
conv_dir([jp([大きさ,N,の構造体配列,A,に,FILE,からﾃﾞｰﾀを読み込む],
  [readfile_struct_array,[FILE,A,N]] ),
  proto([void,readfile_struct_array,
  [['char*',FILE],[struct(A),A],[int,N]] ]),
  funct(readfile_struct_array(FILE,A,N))  ]).                (15)
conv_dir([jp([大きさ,N,の構造体配列,A,において条件,COND,
  を満たす構造体を表示し構造体配列retrにセーブする],
  [retrfile_to_str_ar,[FILE,A,N,COND]] ),
  proto([void,retrfile_to_str_ar,[[struct(A),A],[int,N]] ]),
  funct(retrfile_to_str_parr(A,N)) ]).                       (16)
```

```
conv_dir([jp([大きさ,N,の構造体配列,A,のメンバ,MEM,の和を返す],
  [sum_struct_array,[A,N,MEM]]),
  proto([int,sum_struct_array,[[A,A],[int,N]]]),
  funct(sum_struct_array(A,N)) ]).                                    (17)
```

さて、次に日本語風の呼出文を述語形に変換する方法を簡略化して説明する。いま変数をカスタマイズした呼出文をSTXとし、述語spの引数部に記憶するものとする。このとき呼出文STXは、変換述語conv_dir(CD)を用いて次のような方法（規則節）により述語形に変換される。

```
conv:-sp(STX),conv_dir(CD),member(jp(STX,Y),CD),write(Y),
      add_method(Y).                                                  (18)
add_method(Y):-method_list(L),append(L,[Y],NL),
                ra(method_list(L),method_list(NL)).                   (19)
ra(X,Y):-retract(X),assert(Y).                                        (20)
```

ただし、retract(X)はデータベースからXという述語を消す述語,assert(Y)はデータベースにYという述語を新たに追加する述語である。

このようにして述語method_list(L)の引数部は呼出文STXの述語形Yが加わり、述語形の設計文書が作成されていく。

5．結　言

汎用的なモジュールライブラリを用意しておき、構造体データの構造を与えることで、目的のＣのソースコードを作成する方法について述べた。プロログのリストを用いることで、あらゆるデータ構造に対して一意的にデータ構造の定義を行うことができ、また、それらのデータ構造に対応したモジュールを生成することができた。これら一連の操作は対話形式により進めることができ、専門的知識を持たない者でも、ある程度プログラミングを行うことができる。このことは、プログラムの生産性向上のため有効であり、将来のソフトウェア危機に対する一つの手段となりうると思われる。

今後の課題としては、自動作成されたモジュールのメイン部への反映、メニュー部分のさらなる改善、モジュール郡の充実などが挙げられる。

参 考 文 献

1）恐神,西田：概略設計文からのＣソースコードの生成,福井工業大学研究紀要第２５号, pp. 315～322(1995.3)
2）恐神,西田：フレキシブルなモジュールの構築について，情報処理学会第５０回全国大会，pp. 5-219～220(1995.3)
3）恐神,西田：クラスモジュールの一構成法，情報処理学会第５１回全国大会，pp. 5-157～158(1995.9)

(平成７年12月８日受理)