第8章 リスト構造

見本

リスト構造／メモリの動的確保／単方向リスト／
単方向リストの応用例／双方向リスト／環状リスト

C言語ではさまざまなデータ構造を扱うことができます．リスト構造と呼ばれるデータのつながりは，配列の構造に比べ，追加や削除が容易であり非常に便利なものです．木の構造をしたリストというものもありますが，この章ではつながりが一連になった線形リストについて解説します．具体的にはつながりが一方方向な単方向と，相互につながりをもつ双方向リスト，環状リストなどです．リスト構造を扱ううえで必要な，追加・削除の方法，つなぎ変えの方法などについて解説します．リストの処理ではポインタを多用しますので，ポインタの復習にもなります．

8.1 リスト構造

配列の構造では扱いにくい，データの加除にも柔軟に対応でき，メモリの無駄使いもない，リスト構造と，その操作方法について解説します．

8.1.1 リスト構造とは

一つのデータがあり，次のデータはどこに，そしてその次はどこに，とたどって一連のデータ群を構成する手法です．このように次々と情報をたどっていく仕組みをリスト構造といいます．次のあるいは直前の情報とのつながりをもっているリスト構造はリンクによるリスト(linked list)とも呼ばれます．情報同士を継ぎ手(link)が結んで，鎖のような構造をしています．データとポインタを含む情報部分を節(node)と呼びます．このイメージを**図8.1.1**に示します．

配列の構造はあらじめ大きさが決まっていました．文字列ならば何文字まで格納できる，あるいは何件の情報まで格納できるという枠をあらかじめ用意しておかなければなりませんでした．ですから，数バイトしかない文字列でも，全体を見たときに数10バイトのこともあるというのであれば，その最大値に合わせた枠を用意しなければなりません．配列には長さ固定，個数も固定という制限があります．リスト構造によるデータは，

① サイズが固定でなく，データの大きさに柔軟に対応できる
② ポインタの付け替えだけで並び換えが可能
③ データの挿入や削除を行っても，他のデータを動かす必要がない

などの特徴があります．規模やサイズなどの内容の予

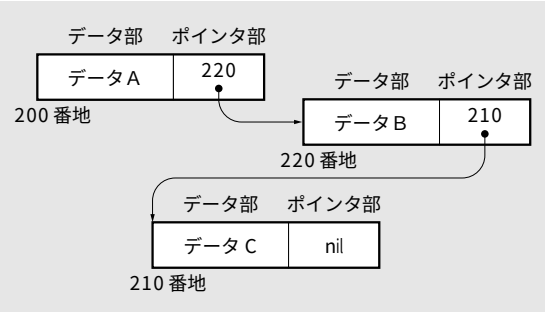

図8.1.1　リスト構造

測がつきにくいようなデータや，並び替えを頻繁に行うようなデータの処理に向いているデータ構造です．配列の欠点を補う手法であるともいえます．

また，リストは自分自身でデータのつながりをもっていることから，配列のように整然と並んだものでなくてもかまいません．つまり，物理的にはあちらこちらに散在しているデータでも，関連付けをすることによって，一連の順序性のあるデータとして扱うことができます．**図8.1.1**では先頭のデータが200番地に，次のデータは220番地にあります．データA内にあるポインタ部が次の場所を指し示しています．データBでは，次の場所情報は210番地を指しているので，戻って210番地が3番目のデータということになります．

そしてその一部が削除されても，その部分だけを，連結から外すことができます．しかも外したその領域のメモリを開放して，他のアプリケーションで利用できるようにすることもできます．そういう意味では，必要なだけのメモリを利用し，無駄なメモリを消費しない手法ということもできます．

8.1.2　自己参照構造体

リスト構造を実現するためには，構造体を利用します．構造体のメンバの中に，自分と同じ形のデータを指し示すポインタのメンバをもちます．このように自分自身の型へのポインタのメンバをもつ形式の構造体を自己参照構造体（self-referential structure）と呼びます．

具体的には，次のように定義します．ここでは簡素化のために構造体内のデータは整数型で1個としました．

```
typedef struct node {
        int data;                   /* データ       */
        struct node *point;         /* ポインタ     */
} NODE;
```

ここで，dataは実際のデータ，pointは次に続くデータへのポインタです．node型の構造体へのポインタであることを明示するために，struct node * と定義しています．nodeを構造体タグと呼ぶことはすでに構造体の項で解説済みです．ここでは同時にNODEという構造体型も定義しています．このように自己参照構造体とは実際のデータと，リンク先への情報とを一緒に構成した構造体のことです．この構造体単位をセル（cell）と呼ぶこともあります．このセルが1個の節（node）を構成します．

メンバとなっているポインタに，リンク先のアドレスを設定し，次はどこへと指示することにより，次々とつながり，一連のデータ群が構成されます．

■（構造体ポインタ変数）->（構造体メンバ）

構造体変数のメンバ指定には，

（構造体変数名）.（メンバ名）

と，ピリオドで区切って表記しました．ピリオドのことを直接メンバ演算子といいます．構造体の指定がポインタ変数で表されているときには，

（構造体ポインタ変数名）->（メンバ名）

という表記で表します．演算子のマイナスと大小記号は数式の中では続いて表記することはありませんので，それを利用しています．これは間接メンバ演算子です．詳細は7.2節を参照してください．

具体的には，構造体へのポインタ変数を，

```
struct node *sp1, *sp2;
```

のように定義したとき，そのメンバのデータは，

```
sp1->data
```

で表され，アドレスへの代入は，

```
sp1->point = sp2;
```

などと表記します．

8.2　メモリの動的確保

プログラム作成の段階から，変数や配列を用意してデータの格納や作業用に利用するということはよくあります．このような領域は一般にメモリの固定の領域に確保されています．その方法ではなく，プログラムが走行中に，プログラム自身の指示でメモリの領域を確保する操作をメモリの動的確保，あるいは動的メモリ割り付け（dynamic memory allocation）といいます．

他の言語でもこの機能は用意されていますが，C言語では`malloc()`という関数が用意されており，データの記憶や作業領域として動的に確保ができます．そして使用しなくなった領域は`free()`関数で開放する，という手法が使われます．C++では`new`や`delete`という演算子を使ってこの操作を行います．

`malloc()`関数の引数は，確保する領域の大きさです．また戻り値は`size_t`型のポインタです．前述の構造体のメモリ領域を確保するには，

```
sp2 = (NODE *)malloc ( sizeof (NODE));
```

などとします．

`size_t`型というのは，sizeof operator type のことで，`sizeof`演算子の演算結果を示す型のことです．実際には符号無しlong型ですので，`NODE`型へのポインタとするために，キャスト演算子を使って代入しています．言葉で説明すれば，`NODE`型の大きさ分の領域を確保しなさいということです．`sp2`には確保された領域の先頭アドレスが代入されます．実際にはぴったりの数のメモリを確保するわけではなく，OSが管理する最小単位ごとのブロックで，16バイト単位などの数のメモリが確保されます．管理最小単位のことをクラスタ・サイズ（cluster size）などと呼ぶこともあります．

動的に確保したメモリ領域は，使用終了後開放するのが原則です．開放しないと，いつまでもメモリを占有し，場合によってはシステム全体で使用できるメモリ領域を蝕んでいくことになります．こうしてメモリ領域がなくなっていくことをメモリ・リーク（memory leak）などと呼びます．開放には，`free()`関数を利用します．引数は確保したときの戻り値，つまりアドレスです．

```
free( sp2 );
```

などとします．

8.3　単方向リスト

図8.3.1に示すように，一方方向にだけリンクされたリストの構造を単方向リスト（singly linked list）と呼びます．1次元リスト，片方向リストという名称も使われます．データを構造体にして，データそのものと，次のデータへのポインタを持っています．`head`は先頭データの場所を示している，やはり構造体へのポインタです．

リストの構造は，配列などと違い，動的にデータの領域を確保していくことができます．あらかじめデータの量が分からないときには配列を確保することはできませんので，リストの構造にするのが便利です．そういう意味で，配列の欠点を補う方法であることは前述のとおりです．

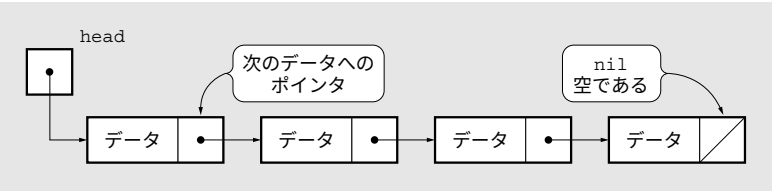

図8.3.1　単方向リスト構造

8.3.1　入力順のリスト

まず，入力された順に，リストを構成していく方法から考えます．

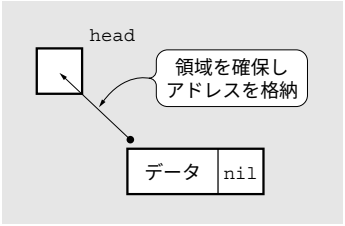

図8.3.2　節のポインタを設定

例題8.3.1 入力されたデータを，入力順にリストに生成し格納する．

最初のデータが入力されたら，**図8.3.2**に示すように，それを格納する領域を確保し，データを格納します．そしてその領域のアドレス，つまりポインタをheadに確保しておきます．これでheadを見れば，先頭のデータ位置が分かることになります．まだデータは1個ですので，次に続くデータはありません．そこでいま格納したデータと同じ構造体の中にある，次のデータへのポインタはNULLを設定しておきます．具体的には，

```
head = (NODE *)malloc ( sizeof(NODE) ); /* 領域確保          */
head->data = (入力されたデータ)          /* データ設定        */
head->next = NULL;                      /* 次のデータは未だない */
```

です．

NULLというのは，正式にはNULLポインタと呼びます．空っぽのポインタということで，どこも指し示していないという意味です．<stdio.h>に定義されているポインタ型の定数です．具体的には数値の0です．説明では同義語でnilという表現を使うこともあります．

そして次からのデータに備え，現在の位置を覚えておきます．

```
prev = head;                            /* 現在位置          */
```

次のデータが入力されると，再度格納する領域を確保し，データを格納します．そして前のデータ構造の中にあるポインタにそのアドレスを登録します．こうして先頭のデータと2番目のデータのリンクが確立します．そして2番目のデータ構造中のポインタには，まだ後がないということでNULLを設定しておきます．

```
new = (NODE *)malloc ( sizeof(NODE) );  /* 次の領域確保      */
new->data = (入力されたデータ);          /* データ設定        */
new->next = NULL;                       /* 次のデータは未だない */
prev->next = new;                       /* 現在位置更新      */
```

などです．この手順を繰り返すことにより，入力順の単方向リストが生成できます．

■ プログラム･リスト

以上の手順をプログラムにします．リストを**リスト8.3.1**に示します．そして一連のリストを表示する関数print_list()も含まれています．プログラムはもう少し単純にできるのですが，ここではわかりやすくするために，先頭のデータ入力と，続くデータの入力の部分を分けて作成しました．

最初のデータが入力されたら，領域を確保します．その領域のアドレスはheadに収納されています．その領域にデータを格納し，次へのポインタにはNULLを設定しておきます．これで，headを先頭指標とした1個のリンクが完成しています．どこまでデータが格納されているかを示すためにprevというポインタ変数も用意しました．この変数は次のデータが入力されたときに，リンク･アドレス設定に利用されます．

2個目以降も同様で，データが入力されるたびに，メ

```
先頭のデータを入力して下さい > 1
&head = 22feec    head = a040c70    data=  1
新しいデータを入力して下さい > 3
prev= 0xa040c70   curr= 0xa040c80   data=  3
新しいデータを入力して下さい > 5
prev= 0xa040c80   curr= 0xa040c90   data=  5
新しいデータを入力して下さい > 4
prev= 0xa040c90   curr= 0xa040ca0   data=  4
新しいデータを入力して下さい > 2
prev= 0xa040ca0   curr= 0xa040cb0   data=  2
新しいデータを入力して下さい > ^D

登録されているデータは
pointer= 0xa040c70 next= 0xa040c80  data=  1
pointer= 0xa040c80 next= 0xa040c90  data=  3
pointer= 0xa040c90 next= 0xa040ca0  data=  5
pointer= 0xa040ca0 next= 0xa040cb0  data=  4
pointer= 0xa040cb0 next= 0x0        data=  2
```

図8.3.3　単方向リスト生成の結果