プログラミングおよび実習II・第 14 回

2004 年 1 月 7 日

1　プログラムの分割コンパイル

前回の演習問題 prog13-1.c では、1 つのソースファイル中に、hashadd、hashsearch、main という 3 つの関数の定義を書きました。これでもまだ 100 行に満たないような小さなプログラムでしかありませんが、実用規模のソフトウェア (プログラム) になってくると、 10000 行、あるいは 100000 行を越えてしまうようなことも珍しくありません。その長いプログラムの全体が 1 つのソースファイルに書かれてしまっていると、次のような問題が発生します。

1. 使用しているコンパイラが対応可能なプログラムの規模を越えてしまい、コンパイルそのものが不可能になってしまうことがある。
2. たとえ、コンパイルが可能だとしても、そのプログラムをコンパイルするだけで、 1 時間、 10 時間、あるいは 1 日間というような長い時間が必要となってくる。
3. プログラムのデバッグ (プログラム中の間違いを訂正すること) を行う際に、例えば 100000 行のプログラム中のたった 1 行を変更しただけでも、もう 1 度その 100000 行のプログラム全体をコンパイルし直さなければならない。
4. プログラムが 1 つのファイルに保存されているので、複数のプログラマの協力によるプログラム開発が困難になる。

これでは、大規模プログラムの開発は、現実的にはほとんど不可能ということになってしまいます。このような問題を解決するため、多くのプログラミング言語には、複数のソースファイルに分割して記述されたプログラムをそれぞれ別々にコンパイルした結果を保存しておき、その後、すべてのコンパイル結果を結合して、最終的に (機械語プログラムとして実行可能な)1 つのオブジェクトファイルを作成するという機能が備わっています。例えば、最終的に myprog という実行可能形式のファイルを作りたい場合に、このソフトウェアのプログラムを、main.c、 sub1.c、 sub2.c という 3 つのソースファイルに分割して記述し、これらをそれぞれコンパイルして、その結果を機械語プログラムのファイル (**オブジェクトファイル**) として保存しておきます。その後、この 3 つの機械語プログラムのファイルを結合 (**リンク**) して、myprog という実行可能形式のファイルを作ります。このような方法でソフトウェアを開発していくと、例えば sub2.c の一部を変更した際には、

1. sub2.c をコンパイルし直して (再コンパイル)、この sub2.c のコンパイル結果であるオブジェクトファイルを作り直す。
2. こうしてできたオブジェクトファイルを、他のオブジェクトファイル (main.c と sub1.c のそれぞれのコンパイル結果) とリンクし直して (再リンク)、myprog を作り直す。

という 2 つの作業を行えばよく、変更されていない main.c や sub1.c の再コンパイルを行なう必要ありません。

このように、1 つのソフトウェアのプログラムを、いくつかのソースファイルに分割し、それぞれを独立にコンパイルすることを**分割コンパイル**と呼びます。実用的なソフトウェアのほとんどすべてが、この分割コンパイルによって開発されています。

1.1 分割する際の方針

では、どのような方針でソースプログラムを分割したらよいのでしょうか。単に分割しさえすればよいというわけではありません。以下のような点に注意して分割する必要があります。

1. それぞれのソースファイル中で定義されている変数や関数がまとまりを持っていること。例えば、そのソフトウエアのある機能に関わっているものがそのソースファイルにまとめられているとか、あるいは、そのソフトウエアで使われるあるデータ構造に関わっているものがまとめられているとかです。ソフトウェアに修正を加える際に、変更しなければならない変数や関数の定義が、あちこちのソースファイルに散在していたのでは大変です。

2. それぞれソースファイルが長すぎないこと。長いソースファイルはコンパイルに時間がかかりますし、それを保守する (例えば、複数のプログラマの協力による開発する) のも困難になってしまいます。100 行程度に収まればそれに越したことはありませんが、先にあげた「まとまり」という観点から考えると、いくらでも短くできるというわけでもありません。長くてもせいぜい 1000 行程度には収めたいものですが、「まとまり」と「短さ」のバランスが重要になります。

1.2 分割コンパイルの例題

前々回作成した prog12-1.c 中のプログラムをソースファイルを、次のような 3 つのソースファイルに分割し、これらを分割コンパイルして、リンクすることで、実行可能形式のファイル prog14-1 を作ってみます。

main.c	……	関数 main の定義を含むソースファイル
qsort.c	……	関数 quicksort の定義を含むソースファイル
bsearch.c	……	関数 binarysearch の定義を含むソースファイル

関数や変数の宣言　関数 quicksort や関数 binarysearch の定義では、printf や exit などの標準ライブラリ関数は使用していないので、prog12-1.c の冒頭にあった

```
#include <stdio.h>
#include <stdlib.h>
```

という 2 行は、qsort.c や bsearch.c では必要なくなります。

C では、プログラム中で関数を使う (呼び出す) 前に、必ずその関数を定義または宣言しておかねばなりません。関数 main の定義の中では、関数 quicksort や関数 binarysearch が呼び出されているので、ソースファイル main.c の冒頭 (関数 main の定義の前であればよい) に、

```
extern void quicksort(ZipInfo *d[], int h, int t);
extern int binarysearch(ZipInfo *data[], int num, int code);
```

の 2 行が必要となります[1]。仮引数名を省略して、

```
extern void quicksort(ZipInfo *[], int, int);
extern int binarysearch(ZipInfo *[], int, int);
```

のように宣言しても構いません。

[1] extern は省略することもできる。

ヘッダファイル　上の 2 つの関数の宣言を、例えば prog14-1.h というファイル中に書いておいて、このファイルを #include を使って 3 つのソースファイル main.c、qsort.c、bsearch.c それぞれに取り込むのも良い方法です。prog14-1.h には、次のように、関数宣言の中に現れる ZipInfo 型の定義や、そこに使われているマクロ定義なども含めるようにします。

prog14-1.h

```
#define PREF_NAME_LEN   (20)
#define CITY_NAME_LEN   (40)
#define AREA_NAME_LEN   (100)

typedef struct {
    int code;                   /* 郵便番号 */
    char pref[PREF_NAME_LEN];   /* 都道府県名 */
    char city[CITY_NAME_LEN];   /* 市町村名 */
    char area[AREA_NAME_LEN];   /* 町域名 */
} ZipInfo;

extern void quicksort(ZipInfo *d[], int h, int t);
extern int binarysearch(ZipInfo *data[], int num, int code);
```

ソースファイル main.c、qsort.c、bsearch.c の冒頭には、それぞれ (関数の宣言や型の定義を書く代わりに)、

```
#include "prog14-1.h"
```

と書きます。この prog14-1.h のように、他のソースファイルに取り込むことを目的に書かれたファイルを**ヘッダファイル**と呼びます。プログラミング環境によってあらかじめ用意されているヘッダファイルを取り込む際は、

```
#include <stdio.h>
```

のように、ヘッダファイル名を <> で囲みますが、自分で作ったヘッダファイルの場合は "" で囲むようにしましょう。こうすることで、システムがあらかじめ用意しているヘッダファイルではなく (cc を実行している) カレントディレクトリに置かれているファイルが探されるようになります。

通常、ヘッダファイルには、

- マクロの定義
- 構造体やデータ型の定義
- 変数や関数の宣言

など、複数のソースファイルで共有される情報を書きます。この例では、quicksort や binaryserach の関数宣言は、qsort.c や bsearch.c では、本来必要ないわけですが、これらの関数宣言をこれらのソースファイルにも #include しておくことで、ヘッダファイル中の宣言とソースファイル中の定義が整合していることを検証することができます。もし整合していなかった場合には、プログラムのコンパイル時に警告 (あるいはエラーメッセージが) が表示されるはずです。

コンパイルとリンク　これら 3 つのソースファイルをコンパイルし、コンパイルしてできたオブジェクトファイルのリンク (結合) するためには、次のようなコマンドを順に実行します。

```
cc -c main.c
cc -c qsort.c
cc -c bsearch.c
cc -o prog14-1 main.o qsort.o bsearch.o
```

最初の 3 行の「-c」は、ccコマンドに対し、その引数となっているソースファイルをコンパイルし、オブジェクトファイルの作成のみを行なうように指示するオプションです。これら 3 行を実行すると、それぞれ main.o、qsort.o、bsearch.o という名前のオブジェクトファイルが作成されます。最後の 1 行では、この 3 つのオブジェクトファイルをリンク (結合) し、その結果を prog14-1 という実行可能なオブジェクトファイルとして保存しています。最後の行は

```
cc main.o qsort.o bsearch.o -o prog14-1
```

のように「-o prog14-1」をコマンド行の末尾に書いても同じです。

これら 4 行のコマンドを実行する代りに、

```
cc -o prog14-1 main.c qsort.c bsearch.c
```

の 1 行を実行しても同じことができますが、この場合 (前回コンパイルして以来変更されていないものを含めて) すべてのソースファイルをコンパイルしてしまうので、一部のソースファイルを修正して再コンパイルする場合は無駄な処理をしてしまうことになります。

2 make コマンド

プログラム開発の過程で分割コンパイルを利用すると、再コンパイルの必要なソースファイルだけをコンパイルすれば済むようになりますが、前回のコンパイル以来、自分がどのソースファイルに変更を加えたかを覚えておき、正確にそのようなファイルだけを再コンパイルすることは現実的にはなかなか困難です。Unix 系 OS 上のプログラミング環境では、この作業を助けてくれる make というコマンドが用意されていることが多くあります。make コマンドによって、再コンパイルの必要なソースファイルだけを自動的に見つけ出してコンパイルし、結果のオブジェクトファイルを再リンクして実行可能形式のファイルを自動的に再構築することが可能となります。

make コマンドを利用するためには、Makefile (あるいは makefile) という名前のファイルを用意し、この中に、どのようなソースファイル (あるいはオブジェクトファイル) によってそのプログラム全体が構成されるのかを記述しておく必要があります。この Makefile の書き方は、C プログラムの書き方とは全く異るので混同しないようにしてください[2]。

prog14-1 を例にとると、この Makefile というファイルには、

prog14-1 のための Makefile

```
1 #
2 #       prog14.1 のための Makefile
3 #
4 PROGRAM = prog14-1
5 OBJS = main.o bsearch.o qsort.o
6 LIBS =
7
8 $(PROGRAM): $(OBJS)
9         cc -o $(PROGRAM) $(OBJS) $(LIBS)
```

[2] Makefile は C プログラムの一部ではありません。

のように書いておきます[3]。ただし、9行目の「cc $(OBJS) ...」の前には間隔文字 (スペース) が並んでいるのではなく、タブ文字が1つ置かれていますので注意してください。

ソースファイルを作ったディレクトリに、このような内容の Makefile を置いておき、そのディレクトリで make コマンド (引数は必要ない) を実行すると、make コマンドは、再コンパイルの必要なソースファイルだけを自動的に見つけ出してコンパイルし、結果のオブジェクトファイルを再リンクして実行可能形式のファイルを自動的に再構築してくれます。

Makefile の # で始る行 (1行目から3行目) は注釈 (コメント) 行です。make コマンドは、Makefile 中の # から行末までを無視します。4行目から7行目までは、この Makefile の中で使われるマクロ (変数) 定義[4]として働きます。これらのマクロ定義は、以下の部分で $(PROGRAM) のような形式で参照することができます。例えば、8行目と9行目は、4行目から6行目のマクロ定義の効果によって、

```
prog14-1: main.o bsearch.o qsort.o
        cc -o prog14-1 main.o bsearch.o qsort.o
```

を意味することになります。

この部分の記述は (8行目の「:」の左に書いた) prog14-1 というファイルの内容は (「:」の右に書かれた) main.o bsearch.o qsort.o の3つのファイルに依存しており、もしこれらの3つのいずれかの内容が変更された場合は (9行目にタブ文字に続いて書かれている)

```
        cc -o prog14-1 main.o bsearch.o qsort.o
```

というコマンドを実行して、prog14-1 を作り直さなければならないということを意味しています。

make コマンドは、prog14-1 の最終更新時刻を main.o、bsearch.o、qsort.o の3つのそれと比較し、そのいずれかよりも prog14-1 が古い場合に「cc -o prog14-1 main.o bsearch.o qsort.o」を実行して prog14-1 を新しくします。ファイルを更新する (作り直す) ときに実行するコマンドを指定する行は、タブ文字で始めなければならないことに注意してください。

この例の LIBS の定義は空となっていますが、特定のライブラリをプログラムと一緒にリンクする場合には、ここに cc コマンドに対するオプションを書いておきます。例えば、sqrt や sin、log 等の標準数学ライブラリ中の関数を使用する際には、

```
LIBS = -lm
```

のように定義しておくことで、9行目で実行される cc のコマンド行の末尾に「-lm」というオプションを追加することができます。

あらかじめ用意されている記述　先の Makefile の記述例には、3つのオブジェクトファイルをリンクして prog14-1 というファイルを作成する方法についての記述はありますが、どのソースファイルをどのようにコンパイルすれば、リンクに必要なオブジェクトファイル (main.o、bsearch.o、qsort.o) を作成できるのかについての記述はありません。これは、次の2行のような記述があらかじめ make コマンドに組み込まれているためです。

```
.c.o:
```

[3]行頭の数は説明のために付した行番号であって、Makefile の内容の一部ではありません。

[4]C のソースプログラム中での #define によるマクロ定義とは全く別のものですが、Makefile 中で同様の働きをします。

```
	$(CC) $(CFLAGS) -c $*.c
```

この 2 行の記述は、「.c」で終る名前を持つファイル (すなわち C のソースファイル) から、「.o」で終る名前を持つファイル (すなわちオブジェクトファイル) を、「$(CC) $(CFLAGS) -c $*.c」というコマンドを実行することによって作成できるということを意味しています。ここで使われている「$*」は特殊なマクロ変数であり、今、作ろうとしているファイルの名前から、「.o」などの拡張子を除いた文字列に展開されます。例えば qsort.o を作ろうとしている場合、「$*」は「qsort」という文字列に展開されます。また、$(CC) や $(CFLAGS) というマクロ変数は、あらかじめ

```
CC = cc
CFLAGS =
```

のように定義されています。Makefile の冒頭で、これら CC や CFLAGS を自分で再定義することもできます。例えば、先の Makefile の冒頭に、

```
CFLAGS = -O
```

の行を追加すると、それぞれのソースファイルをコンパイルする際に、cc コマンドに -O というオプションが渡されるようになります。

ヘッダファイルへの依存関係の記述　ヘッダファイルを他のソースファイルに #include している場合には、ヘッダファイルを変更したら、そのファイルを #include しているソースファイルを再コンパイルしなければなりません。この再コンパイルを make コマンドに自動的させるためには、この依存関係を Makefile 中に記述しておく必要があります。例えば、前節で解説したように、qsort や bsearch の関数宣言を prog14-1.h という名前のヘッダファイルに書き、これを各ソースファイルに #include した場合、Makefile の内容は次のようにしておきます。

prog14-1.h への依存関係を記述した Makefile

```
#
#       prog14-1 のための Makefile
#
PROGRAM = prog14-1
OBJS = main.o bsearch.o qsort.o
LIBS =

$(PROGRAM): $(OBJS)
        cc $(OBJS) -o $(PROGRAM) $(LIBS)

$(OBJS): prog14-1.h
```

最終行の記述により、各オブジェクトファイルの最終更新時刻 (前回コンパイルした時間) がヘッダファイル prog14-1.h のそれよりも古い場合にも、対応するソースファイルが再コンパイルされるようになります。

3　演習問題

次の演習問題に取り組みなさい。 prog14-1 については、完成したら「mprog2 prog14-1」を実行して提出してください。

3.1 prog14-1

まず、Prog2 ディレクトリに prog14-1 という名前のディレクトリを新しく作成しなさい。次に、このディレクトリに、今回解説した要領で main.c、bsearch.c、qsort.c、prog14-1.h、Makefile の 5 つのファイルを作成し、ここで make コマンドを実行するだけで (前々回の演習問題の prog12-1 と同じ機能を持つ) オブジェクトプログラム prog14-1 が作られるようにしなさい。

3.2 prog14-2

余裕のある受講者は、同様に prog14-2 という名前のディレクトリを作成し、前回の演習問題で作成した prog13-1.c を、main.c、hash.c、prog14-2.h の 3 つのファイルに分割したものを置きなさい。hash.c には、hashadd と hashsearch の 2 つの関数定義が含まれるようにいしなさい。また、これらのファイルから prog14-2 という実行可能なオブジェクトファイルを構築するための Makefile を用意しなさい。

プログラミングおよび実習Ⅱ・第 14 回・終り

付録 A. ポインタ型について

変数や配列とメモリ　プログラム中の変数や配列 (の各要素) に記憶される各種のデータは、実際にはコンピューターの中のメモリと呼ばれる記憶装置に格納されています。1 つの C プログラムから利用することのできるコンピュータのメモリは、長い紙テープのようなものに見えます。この紙テープは、多くの小さな欄 (区画) が連なった形をしており、1 つ 1 つの欄には、それぞれ (たとえば) 8bit (1byte) 長の情報を記憶することができます[5]。あるシステムでの `int` 型のデータが 32bit (4byte) 長であったとすると、このシステムの `int` 型の変数の値は、紙テープ上の (連続した)4 つの欄を使って記憶されます。また、`int` 型の要素 100 個からなる配列の内容は、(連続した)400 個の欄を使って記憶されます。変数や配列要素の値を代入演算子などで変更するということは、紙テープのいくつかの欄に記憶されている情報を書き換えるということに対応します。

アドレスとメモリ空間　紙テープ (メモリ) のそれぞれの欄は、テープの始まりから何番目の欄であるかで識別されます。この番号を、その欄の**アドレス** (番地) と呼びます。実行中のプログラムにとって、このアドレスの振られた紙テープのように見えるものを**メモリ空間**と呼びます。

C プログラム中の変数や配列は、このメモリ空間の中から、あるアドレス (番地) から始まる、ある大きさの領域 (いくつかの欄) をそれぞれ割り当てられることで、その値が記憶されるわけです。この領域の先頭の欄のアドレスを、その変数や配列のアドレスと呼びます。プログラムの実行が開始されてから、実行が終了するまで領域が割り当てられたままの変数や配列もあれば[6]、あるブロックの実行が始まった時に割り当てられ、そのブロックの実行が終了した時には解放されるものもあります[7]。

左辺値とアドレス演算子、ポインタ型　C プログラム中で、変数や配列要素など、代入演算子 `=` の左辺に書くことのできる式を一般に**左辺値**と呼びます[8]。左辺値は、何らかのデータを格納する働きを持つものを指し示す式と言うことができます。たとえば、`x` と `y` が `int` 型の変数であった場合、

```
x = y + 1;
```

という文を考えると、`=` の右辺に現れている変数 `y` は、`y` という変数に格納 (記憶) されている (`int` 型の) 値を表わしていますが、`=` の左辺に現れている変数 `x` は、`x` に格納されている値を表わしているのではなく、値の格納場所としての変数 `x` そのものを指していると考えることができます。このように、同じ変数でも、式のどの位置に書かれているかによって、左辺値として扱われたり扱われなかったりするということに注意が必要です。

式 e が左辺値 (つまり、データの格納場所を表わすことのできる式) であるとき、`&`e という式で、メモリ空間中で e の値を記憶するために割り当てられている領域の先頭アドレスを表わすことができま

---

[5] つまり、1 つの欄に記憶される情報で 256 通りの値を区別することができます。

[6] このような変数や配列を**静的変数** (静的配列) と呼びます。たとえば、C の予約語 `static` を付けて宣言された変数や配列は静的変数 (静的配列) となります。関数定義の外側で宣言される大域変数もそうです。

[7] このような変数や配列を**自動変数** (自動配列) と呼びます。予約語 `static` 無しに宣言された局所変数や局所配列、関数の仮引数は自動変数 (自動配列) となります。

[8] 他にも、たとえば、`x` が構造体型の変数で、`m` がその構造体のメンバ名であった場合、`x.m` という式も左辺値の 1 つとなります。

す[9]。この & は**アドレス演算子**と呼ばれ、左辺値から、そのアドレスを取り出す働きを持ちます。

C プログラム中でのアドレスは、int 型や long 型のような単なる整数型とは異なるデータ型として取り扱われます。たとえば、x が int 型の変数であるとき、&x という式は「int 型へのポインタ型」と呼ばれるデータ型を持ちます。また、 a が double 型の配列であるとき、&a[0] の型は「double 型へのポインタ型」となります[10]。これらの型は int 型や double 型とは全く異なるデータ型であることに注意が必要です。一般に、式 e のデータ型が T あるとき、式 &e のデータ型は「T 型へのポインタ型」となります。型 T が異なれば、T 型へのポインタ型もそれぞれ異なる型となりますので、たとえば、「int 型へのポインタ型」と「double 型へのポインタ型」は明確に区別する必要があります。

ポインタ型の変数　ポインタ型の値 (アドレス) を記憶する変数を利用することもできます。T 型へのポインタ型の値を記憶する変数は、T 型の値を記憶する変数の宣言の変数名の前に * を書いて宣言します。たとえば、int 型へのポインタ型の変数 p は、

```
int *p;
```

のように宣言します。ここで宣言された p という変数は、代入演算子 = を使って、int 型へのポインタ型の値 (アドレス) を格納したり、式中に p と書くことで、そこに格納されている値 (アドレス) を参照することができます。

次のように、int 型の変数 x、y と、ポインタ型の変数 p、q を同時に宣言することもできます。

```
int x, y, *p, *q;

p = &x;
q = &y;
```

変数の宣言の中で、 4 つの変数をどのような順に並べても構いません。* の付いた変数は int 型へのポインタ型の変数として、付いていない変数は int 型の変数として宣言されます。この例では、 4 つの変数の宣言に続いて、変数 p に変数 x のアドレスを、変数 q に変数 y のアドレスを、それぞれ代入しています。

ポインタ型へのポインタ型　型 T へのポインタ型の変数は、T 型へのポインタ型の左辺値となりますので、アドレス演算子 & でその変数のアドレスを取り出すと、その型は「T 型へのポインタ型へのポインタ型」となります。たとえば、上の例のように、変数 p が int 型へのポインタ型であった場合、&p という式は、この変数 p のアドレスを表わしますが、そのデータ型は「int 型へのポインタ型へのポインタ型」となります。

このような、ある型へのポインタ型へのポインタ型の値 (アドレス) を変数に記憶することもできます。その場合、次の例のように、変数の宣言に * がさらに追加されることになります。

```
int x, *p, **pp;

p = &x;
pp = &p;
```

---

[9] &e という式は、e が格納されているアドレスそのものの値を表わしている (そのアドレスがどこかに格納されているわけではない) のですから、この &e という式はもう左辺値ではありません。したがって、 &&e ような形の式は存在しません。

[10] アドレス演算子 & よりも、配列添字演算子 [ ] の方が結合の優先度が高いので、&a[0] は &(a[0]) と書くのと同じ意味になります。

間接参照演算子　型 T の左辺値に対してアドレス演算子 & を適用すると、その左辺値を格納しているメモリ領域の先頭アドレスを、T 型へのポインタ型の値として取り出すことができますが、逆に T 型へのポインタ型の値 (アドレス) を表わす式 e があった場合、$*e$ という式で、アドレス e に格納されている T 型のデータを表わすことができます。この * は**間接参照演算子**と呼ばれます。$*e$ は左辺値となりますので、アドレス e に格納されているデータを読み出すだけでなく、そのデータを書き換えたり、アドレスを取り出したりすることもできます。一般に、式 e が左辺値であった場合、*&e や *&*&e、*&*&*&e ... は、すべて e と同じ意味となることに注意して下さい。たとえば、次のプログラムを実行すると、最後の printf の呼び出しでは「x = 15」が出力されます。

```
int x, *p;

*&x = 10;
p = &x;
*p = *p + 5;
printf("x = %d\n", x);
```

int 型へのポインタ型の変数 p の宣言を「int *p;」のように書くのは、この間接参照演算子 * に由来しています。「int *p;」は、「*p が int 型になるような変数 p」の宣言と読むと理解がしやすいと思います。同様に「int **pp;」は、pp に 2 回 * を適用すると int 型の式になるような変数ということです。

ナルポインタ (NULL)　ポインタ型の値としての 0 は特別な意味を持っています。いかなる変数や配列も、メモリ空間の 0 番地から始まる領域に割り当てられることはありません。C のプログラム中では、ポインタ型の 0 を「何も指していない」ということを表わすための定数として利用します。stdio.h や stdlib.h などの標準ヘッダファイルには、このポインタ型の 0 が NULL というマクロとして定義されていますので、通常 C プログラム中では、これらのヘッダファイルを #include しておいて、この NULL というマクロを使用します。ポインタ型の 0 に対して間接参照演算子 * を適用して、そこに置かれたデータにアクセスすることはできません。

ポインタ型の仮引数と返り値　ポインタ型の値は、関数の引数として渡したり、逆に関数の返り値として呼び出し元に返したりすることもできます。このような関数定義での仮引数の宣言は、変数の宣言の場合と同様に、仮引数名の前に * を追加して行ないます。また、返り値がポインタ型であることを示すためには、関数名の前に * を追加します。次のプログラムは、double 型へのポインタ型の引数 p、q を受け取ると、それぞれが指す (double 型の) データの大きさを比較して、大きい方が置かれているアドレスを返り値として返すような関数 which の定義です。2 つのデータが等しかった場合には NULL (ポインタ型の 0) を返り値として戻しています。

```
#include <stdlib.h>

double *which(double *p, double *q)
{
    if (*p > *q) return p;
    if (*p < *q) return q;
    return NULL;
}
```

付録 B. 前回の実力チェックの解答例

dsearch.c

```
 1 #include <stdlib.h>
 2
 3 double *dsearch(double d[], int n, double x, int start)
 4 {
 5     int i;
 6
 7     for (i = 0; i < n; i++) {
 8         if (d[start] == x)
 9             return &d[start];
10         start = (start+1)%n;
11     }
12     return NULL;
13 }
```

付録 C. 前回の演習問題の解答例

prog13-1.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 #define FILE_NAME       "/home/sample/mprog2/zipcode"
 5
 6 #define MAX_DATA        (200000)
 7 #define PREF_NAME_LEN   (20)
 8 #define CITY_NAME_LEN   (40)
 9 #define AREA_NAME_LEN   (100)
10
11 typedef struct {
12     int code;                    /* 郵便番号 */
13     char pref[PREF_NAME_LEN];    /* 都道府県名 */
14     char city[CITY_NAME_LEN];    /* 市町村名 */
15     char area[AREA_NAME_LEN];    /* 町域名 */
16 } ZipInfo;
17
18 #define HASH_TABLE_SIZE (3*MAX_DATA/2)
19 #define HASH(c, s) ((c)*43%(s))
20
21 int hashadd(ZipInfo *htable[], int tsize, ZipInfo *p)
22 {
23     int n = tsize;
24     int h = HASH(p->code, tsize);
25
26     while (n-- > 0) {
27         if (htable[h] == NULL) {
28             htable[h] = p;
29             return h;
30         }
31         h = (h+1) % tsize;
32     }
33     return -1;
34 }
35
36 ZipInfo *hashsearch(ZipInfo *htable[], int tsize, int code)
```

```
37 {
38     int n = tsize;
39     int h = HASH(code, tsize);
40     ZipInfo *p;
41
42     while (n-- > 0) {
43         p = htable[h];
44         if (p == NULL || p->code == code)
45             return p;
46         h = (h+1) % tsize;
47     }
48     return NULL;
49 }
50
51 int main()
52 {
53     static ZipInfo data[MAX_DATA], *hashtable[HASH_TABLE_SIZE];
54     FILE *fp;
55     int code;
56     int num;
57     ZipInfo *p;
58
59     fp = fopen(FILE_NAME, "r");
60     if (fp == NULL) {
61         printf("%s というファイルが読めません\n", FILE_NAME);
62         exit(EXIT_FAILURE);
63     }
64
65     num = 0;
66     while (fscanf(fp, "%d %s %s %s", &data[num].code,
67                   data[num].pref, data[num].city, data[num].area) == 4) {
68         if (hashadd(hashtable, HASH_TABLE_SIZE, &data[num]) < 0) {
69             printf("ハッシュ表が溢れました\n");
70             fclose(fp);
71             exit(EXIT_FAILURE);
72         }
73         num++;
74     }
75
76     fclose(fp);
77     printf("%s から %d 件のデータを読み込みました\n", FILE_NAME, num);
78
79     while (1) {
80         printf ("郵便番号を入力してください : ");
81         if (scanf("%d", &code) != 1)
82             break;
83         p = hashsearch(hashtable, HASH_TABLE_SIZE, code);
84         if (p == NULL)
85             printf("対応する地域がありません。\n");
86         else
87             printf("%07d %s %s %s\n", p->code, p->pref, p->city, p->area);
88     }
89     printf("検索を終了します。\n");
90     return EXIT_SUCCESS;
91 }
```