第5章
文字列の操作

C言語の文字列／文字列の初期化と代入／文字列の処理／
便利な関数を作る／文字列処理の応用／コマンドラインの引数／
コマンドライン引数の取り扱い／引数の応用／main関数の戻り値

C言語における文字列の処理，操作には特徴があります．BASIC言語を経験された方には，少々不可解な面があるかもしれません．BASIC言語では数値を格納する変数と同じように，文字列を格納する文字変数というものもありました．xxx$というものです．処理の具体的な方法は違っても，ごく簡単に文字列が扱えました．

C言語では文字列を扱うためには，ポインタに関する知識や配列に対する知識が必要です．この章では，ポインタや配列の復習を兼ねて，文字列処理の関数を作ってみます．これらの多くの関数は同等のものが，<string.h>というヘッダ・ファイルに定義されている文字列処理関数として，標準ライブラリに含まれています．無駄な仕事が多いのですが，ポインタや配列を，より深く理解するために有効と思われますので，あえて無駄と思われることをやってみます．文字列の探索・置換については第13章で，文字列のファイル操作については第14章で解説しています．

5.1　C言語の文字列

C言語で文字列（stringまたはcharacter string）と呼ばれるものには一定のルールがあります．それを理解していないと思わぬ失敗をします．

5.1.1　文字列の表現

C言語では文字列を表現するときには，ダブル・クォーテーションでくくって，

```
"abcdefg"
```

のように表現します．

シングル・クォーテーションでくくった，

```
'a'
```

は文字単体を表し，文字列とは明確に区別します．1文字だから，複数だからという以上の違いがあることは2.5節でも解説しました．

シングル・クォーテーションでくくった文字単体は，char型の定数または変数です．つまりメモリ配置上も，1バイトの領域です．しかしダブル・クォーテーションでくくった"a"は，たとえ1文字だけでも，それとは異なります．ダブル・クォーテーションでくくれば，文字列の扱いとなります．メモリも1バイトではなく，2バイト使用します．

5.1.2　文字列の形式

C言語の文字列は，その文字列の終わりを示す終端コード（code：符号）が付加されたものです．終端コードには，

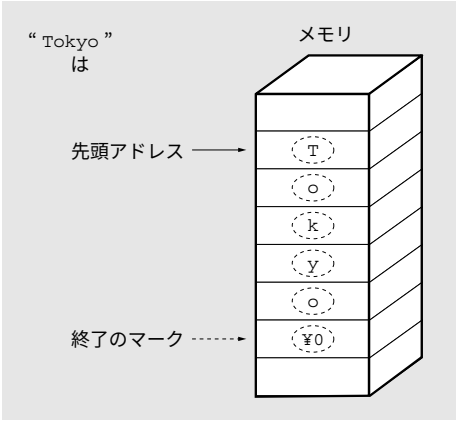

図5.1.1　文字列の格納

文字コードとしては無効な0x00が利用されています．0x00を文字単位表記式に'¥0'というように，エスケープ記号¥と0を使って表します．

たとえば，"Tokyo"という文字列は，メモリに配置されるのは，**図5.1.1**に示すように，Tokyoの末尾に終端コードの'¥0'が付加されています．

5.1.3　文字列の取り扱い

文字列を取り扱う際には，その先頭アドレスと終端コードを目安に行います．「先頭から順に見ていって，終端コードが出てきたら終わりにする」という扱いです．ですから，終端コードのない文字列は，終わりがないわけで，扱うときには暴走ということになりかねません．つまり文字列としての要件はありませんので，文字列とは見なされません．

文字列処理の標準ライブラリもすべて，この前提で作られています．文字列を先頭アドレスと終端コードによって管理する，取り扱うと決めておくことにより，文字列の長さが個別に違っていても，つまり可変長でも，同様の取り扱い方法ができるというメリットがあります．

5.1.4　文字列の値

文字列の値あるいは文字列値(string value)という用語が使われることがあります．蛇足ながら誤解のないように，少々コメントを加えておきます．これはC言語では文字列の内容，中身の値のことではなく，文字列を指しているポインタの値のことです．C言語が前述のように，文字列の表現にはポインタを利用していることの現れです．以上をまとめると，'と'でくくった文字は整数変数と同じ扱いになり，"と"でくくった文字列はポインタとして扱うということです．ポインタ扱いということは以下に解説する配列とも似ているということになります．

5.1.5　文字列と1次元配列

文字列は**図5.1.1**のような配置で格納されます．文字列の取り扱いと1次元の配列の取り扱い，そしてまたポインタによる管理などは，まったく同じものと考えてもよいのです．

実例は後の節で出てきますが，ポインタ変数を

```
char *cp;
```

と定義して，その内容をアクセスするときに，ポインタとして扱うならば，

```
*(cp + i )
```

と指定できますし，配列として扱うならば，

リスト5.1.1　ポインタと1次元配列

```
/* char ポインタと1次元配列  */
#include <stdio.h>

int main ( void )
{
  char *cp = "ABCDEFG";           /* ポインタで文字列定義 */
  char ca[] = "abcdefg";          /* 配列で文字列定義     */

  printf ("ポインタ扱い %c  %c¥n", *(cp+1), *(ca+5));
  printf ("配列扱い     %c  %c¥n", cp[1], ca[5] );
  return 0;
}
```

```
ポインタ扱い B  f
配列扱い     B  f
```

図5.1.2　ポインタと配列の実行結果

cp[i]

と表現することもできます．配列名でも同様です．**リスト5.1.1**に示すプログラムで試してみます．

このプログラムでは，char型のポインタ変数cpを定義し，初期値で大文字のA～Gを与えています．またchar型の配列caには小文字のaからgを設定しています．

続く表示で，1行目では両変数をポインタとして扱い，*(cp+x)の形式で指定しています．2行目では，配列として扱い，cp[x]の形で指定しています．実行結果を**図5.1.2**に示します．両者の結果は同じになっています．これで，ポインタ変数と1次元配列の変数名は同じレベルのものということがわかります．

5.2　文字列の初期化と代入

文字列として用意した器，つまり変数あるいは定数領域に，文字列を格納する際の方法です．

5.2.1　変数宣言時の初期化

文字列の定義･宣言には各種の方法があります．文字列は1次元配列です．配列への一括代入は，変数宣言のときにかぎって行うことができます．初期化あるいは代入の可否を**リスト5.2.1**にまとめて示します．処理の途中で一括代入することはできません．

このとき，配列caは文字列変数というよりは，文字列定数です．確保されている領域に新たに代入されようとした文字列が，格納しきれるかどうか不明なので，途中の代入は禁止されていると考えてもかまいません．

この例では変数宣言時に，配列の大きさを指定せずに，初期値として与えたリテラルに応じて，その大きさの確保をコンパイラに任せています．このような場合には，処理中の代入は，サイズを越えて代入し，次の領域，実際には関数のリターン・アドレスなどが確保されている場所を破壊することもありえますので，非常に危険です．

リスト5.2.1　配列の初期化

```
/* 文字配列の初期化可否        */
#include <stdio.h>

int main ( void )
{
  char  ca1[8] = "abcdefg";             /* 配列変数宣言時初期化 */
  char  ca2[] =  "abcdefg";             /* 要素数は省略可       */
  char  ca3[8];

  printf ( "%s¥n", ca1 );
  printf ( "%s¥n", ca2 );

  char  *cp = "ABCDEFG";                /* ポインタとし定義も可 */

  printf ( "%s¥n", cp );

#if 0
  ca3[] = "abcdefg";                            /* 途中の代入は不可     */
  printf ( "%s¥n", ca3 );

  /*  複数の配列に定義する場合  */
  char  fruit0[5+1] = "apple";          /* 個別の配列に定義     */
  char  fruit1[5+1] = "grape";
  char  fruit2[4+1] = "pear";
  char  fruit3[5+1] = "peach";
  char  fruit4[6+1] = "orange";

                                        /* 2次元配列に定義する */
  char  fruits[5][10] = {"apple","grape","pear","peach","orange"};

                                        /*  ポインタで定義する */
  char  *fruits[5] = {"apple","grape","pear","peach","orange"};
#endif
  return 0;
}
```

5.2.2　処理中の代入

処理中には，**リスト5.2.1**に示したような方法で，配列への一括代入はできません．他の方法によって代入します．一番単純な方法は，配列の1要素ずつ代入する方法です．**リスト5.2.2**に示しますが，数バイトならばよいとして，文字列が長くなった場合には不適です．実行結果を**図5.2.1**に示します．代入した要素だけが入れ替わっています．

文字列の一括代入には，通常strcpy()関数を利用します．この関数と同等の関数を後刻例題で自作もします．同様の処理は，sprintf()関数を利用しても達成できます．いずれも終了を示す'¥0'符号も付加してくれます．これらの関数を利用した文字列代入の例を**リスト5.2.3**に示します．また'¥0'を強制的に付加すれば，memcpy()関数でも同様です．実行結果を**図5.2.2**に示しますが，この場合には定義で確保した文字配列の要素数を絶対に越えないという配慮が必要です．

5.2.3　文字列へのポインタ

文字列は1次元配列と同じ様式で格納されていますので，文字列内の文字をアクセスする場合には，ポインタによる方法を使うことができます．たとえば，

```
char        *cp;
```

とchar型へのポインタ変数を定義し，

```
*cp = "Tokyo";
```

とすることによって，文字列"Tokyo"のアドレスが確保できます．実際の文字列はデータ領域に確保されています．

プログラム中に記述された文字列は定数です．このような定数を文字列リテラル(string reteral)と呼びます．また**リスト5.2.3**に示したstrcpy()やsprintf()に記述された文字列定数についても，文字列はデータ領域に別に定義され，コード領域には，そのアドレスのみが収容されることになります．プログラム(コード)中に，文字列が入るわけではありません．

リスト5.2.2　1要素ずつの代入

```
/* 文字配列の初期化・代入可否  */
#include <stdio.h>

int main ( void )
{
  char ca[] = "abcdefg"; /* 配列変数宣言時初期化 */

  printf ("%s¥n", ca);


  ca[0] = 'A';           /* 要素ごとに代入は可   */
  ca[1] = 'B';
  ca[2] = 'C';
  ca[3] = 'D';
  printf ("%s¥n", ca);
}
```

```
abcdefg
ABCDefg
```

図5.2.1　文字列初期化の実行結果

リスト5.2.3　文字列の一括代入

```
/* 文字配列の初期化・代入可否  */
#include <stdio.h>
#include <string.h>

int main ( void )
{
  char ca[20] = "abcdefg";  /* 配列変数宣言時初期化   */

  printf ("%s¥n", ca);

  strcpy ( ca, "ABCDEFG" ); /* strcpy関数で代入      */
  printf ("%s¥n", ca);

  sprintf ( ca, "1234567" );/* sprintf関数で代入     */
  printf ("%s¥n", ca);

  memcpy ( ca, "abcdefg¥0", 8 ); /* strcpy関数で代入 */
  printf ("%s¥n", ca);
  return 0;
}
```

```
abcdefg
ABCDEFG
1234567
abcdefg
```

図5.2.2　文字列一括代入の実行結果