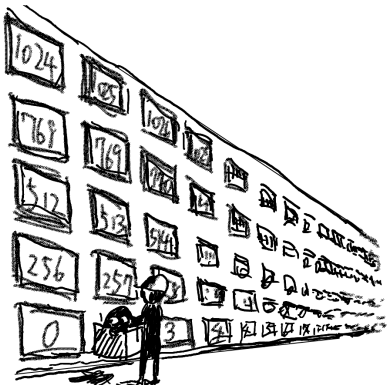
第 9 章

文字列とポインタ



コンピュータ言語の「文字列」とは、複数の「文字」から構成されるもので、長さがあります。10 文字の場合も、100 文字の場合もあるでしょう。長さが 1 文字だと、単なる「文字」と同じに思えるかもしれませんが、コンピュータ言語にとっては、たまたま長さが 1 文字であっただけで、「文字列」と「文字」とは、決定的に異なるものとして扱います。さらに長さが 0 文字だと「空文字列」などと呼んで、文字列ではあっても中身が何もない、というものもよく登場します。

C 言語の文字列は、専用の型があるわけではなく、char 型の配列がその役目を担います。文字列のための特別な文法が少しあって、文字列操作のための標準ライブラリ関数がありますが、基本的に配列なので、お世辞にも便利とは言えません。

とは言え、画面表示などには、やはり必要な機能です。配列操作のよい例題にもなるので、深入りせずに使い方を学びましょう。配列と表裏一体のポインタについても、少し触れます。

キーワード

- ヌル文字
- 文字列リテラル=初期化子
- `<string.h>`
- strlen(), strcpy(), strcat(), strcmp()
- ポインタ=アドレス
- const, *, &
- ポインタの配列

9.1 文字列=char の配列

C 言語の 1 文字は char 型で扱い、文字定数は'A' のようにシングルコーテーションで囲って示しました。ここでは、複数の文字からなる**文字列** (string) を取り上げます。

文字列は、char の並んだものとして扱い、終端を表すために**ヌル文字** (null character) という、特別な文字（文字コードの 0 番）を配置します。一種の**番兵**です。つまり、文字列自体にはヌル文字を含められませんが、それ以外の char の並んだものなら、長さに制限なく扱えます。ヌル文字は、エスケープシーケンスの 8 進数コードを使って'\0' と表します。整数の「0」でも働きは同じですが、文字定数にすることで「文字である」という意図が読み取れます。

文字列は char の配列に格納して扱います。配列要素に'A' のような文字定数を代入できますし、初期化子に文字定数を並べて構いません。

```
/* int配列の初期化 */
int a[4];  a[0]=10;  a[1]=20; ...
int b[4] = { 10, 20, 30, 40 };
int c[]  = { 10, 20, 30, 40 };
```

```
/* char配列の初期化 */     // △
char a[4]; a[0]='T'; a[1]='h'; ...
char b[4] = { 'T','h','e','\0' };
char c[]  = { 'T','h','e','\0' };
```

頻出ミス

「The」の **3 文字の文字列**を格納するのに、char 配列は **4 要素**必要です。終端のヌル文字を忘れがちですので、右上の例の初期化子は**実際には用いません**。

初期化子には、文字列専用の文法があります。ダブルコーテーションで囲った**文字列リテラル**が初期化子のブロックの役目を果たし、しかも終端のヌル文字を含みます。これなら編集も容易で、要素数を省略すれば**ヌル文字分を忘れない**ので、実用的で安全です。

```
/* int配列の初期化 */
// 対応する機能なし
```

```
/* char配列の初期化 */
char d[4] = "The"; // △3ではない
char e[]  = "The"; // ○推奨
```

変数定義の後に代入したくなっても、初期化子ブロックが代入できないのと同じように、文字列リテラルも代入できません。もちろん 1 文字ずつの代入はできますが、実用的ではありません。通常はこの後に紹介する、標準ライブラリの文字列コピー関数を使います。

```
int f[4]; // int配列への代入
// f = { 10, 20, ... };    // NG
f[0] = 10;
f[1] = 20;
f[2] = 30;
f[3] = 40;
```

```
char f[4]; // char配列への代入
// f = "The";    // NG
f[0] = 'T';
f[1] = 'h';
f[2] = 'e';
f[3] = '\0'; // 忘れそう
```

鋭意作成中

9.2 文字列操作の標準ライブラリ関数

文字列 str を画面に表示するには、<stdio.h> の printf("%s", str) あるいは puts(str) を使います。puts() は str を表示した後に、さらに改行します。

ほかにも、<string.h> に次のような標準ライブラリ関数が用意されています。

長さ size_t strlen(const char *s);
コピー char *strcpy(char *dest, const char *src);
連結 char *strcat(char *dest, const char *src);
書式変換[*1] int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
比較 int strcmp(const char *s1, const char *s2);

見慣れない型がいくつかありますが、意味は次の通りです。

size_t sizeof 演算子の返す**符号なし整数型**で、標準ライブラリでは、長さやバイト数を表すときに用いられます。（実体としては unsigned int や unsigned long などのいずれかが、**処理系依存**で選ばれています。）本書では符号なしの型を積極的には用いませんから、strlen() と int 型変数との間で代入や比較の際には、厳密には **int へのキャスト**が必要です。もっとも、C 言語のいい意味のいい加減さのおかげで、警告されない場面も多いです。

char* 「char へのポインタ」です。今は「char の配列」と同じと理解してください。

const char* 「char へのポインタ」ではありますが、配列の中身を書き換えません、という意思表明だと思ってください。const は constant（不変の、一定の）の意味、つまり変数が変化しないことを示す修飾子です[*2]。

次節からは、標準ライブラリを実装してみます。文字列操作の理解を深めましょう。

MEMO: 付録に独立させる

[*1] sprintf() と snprintf()^{C99} は正確には <stdio.h> の関数です。

[*2] const で修飾された変数に代入すると、通常はコンパイルエラーになるでしょう。かつては単に const を読み飛ばすだけのコンパイラも存在しました。

9.2.1　文字列の長さ

strlen() は、文字列の長さを返す標準ライブラリ関数です。この動作を理解するために、自作しなおしてみたのがソースコード 9.1 です。

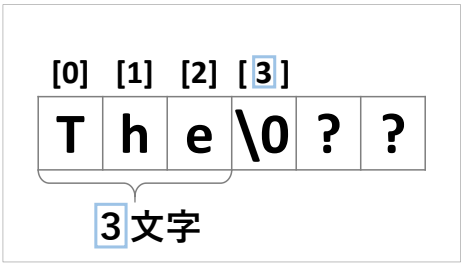
str_length(str) 配列の先頭から'\0' を探しています。str[i] が'\0' になったら、そのときの i が文字列の長さになります。配列の添字が 0 始まりのおかげで、-1 のような調整項が不要で、うまく計算できます。

コラム：ヌル文字の向こうに

文字列を扱う標準ライブラリ関数は、ヌル文字より後の配列要素を参照しません。文字列のデータ構造としては、何が入っていても関係がありません。

ソースコード 9.1　文字列の長さ

```
1  #include <stdio.h>
2  #include <string.h> // strlen()
3
4  /* strlen()の自作関数：文字数を数える */
5  int str_length(char str[]) {
6      int i; // ループ後にも必要なので、ここで定義
7      for (i=0; str[i]!='\0'; i++) {
8          // dummy
9      }
10     return i;
11 }
12
13 void test_str_length(char s[]) {
14     printf("'%s', %d, %d\n",
15            s, (int)strlen(s), str_length(s));
16 }   // (int) でキャストする代わりに %zu で表示してもよい(C99)
17
18 int main(void) {
19     char str[] = "The";
20     printf("%d\n", (int)sizeof(str)); // 4（'\0'を含むバイト数）
21     test_str_length(str);             // 3（文字列の長さ）
22
23     test_str_length("The");  // 3（文字列リテラルを引数に渡せる）
24     test_str_length("");     // 0（長さ0の空文字列）
25     test_str_length("あ");   // 3(UTF-8の場合) または 2(SJISなど)
26     return 0;
27 }
```



test_str_length(s) 標準ライブラリの strlen(s) と、自作の str_length(s) について、比較するために両方の値を表示しています。同じ値が表示されることを確認しましょう。strlen() の型は `size_t` ですので、printf() で表示するには工夫が必要です。実体が unsigned int や unsigined long などの可能性がありますから、書式文字列を`"%d"`や`"%ld"`と決めてしまうと、可搬性が失われます。`size_t` には、C99 で新設された`"%zu"`を使います。あるいは、int にキャストしてしまえば、使い慣れた`"%d"`で大丈夫です。

main() 文字列`"The"`の占めるバイト数 (=4) を表示してから、`"The"`をはじめとする何通りかの文字列に対して test_str_length() を呼び出しています。まずは sizeof と strlen() の食い違いに注意してください。

test_str_length() の引数は char 配列ですが、23–25 行目のように、文字列リテラルを直接与えても大丈夫です。ここからわかるように、**文字列リテラルの型は char 配列**（正確には char へのポインタ）です。変数に結び付けなくても char 配列を作り出す、C 言語に昔からある、特別な文法です。

空文字列 (empty string)（長さが 0 文字の文字列）のような極端な例は、間違いを見つけやすいので、積極的に試しておくべきです。ここでは 24 行目で実施しています。境界値分析（**??**ページのコラム）にも通じる手法です。

頻出ミス

strlen(str) は、sizeof(str) と同じと思いがちですが、`'\0'` を含めずに数えるので、1 小さくなります。コピー先の領域確保の際には、要注意です。

コラム： マルチバイト文字の入った文字列リテラル

ダブルコーテーション（"）に囲まれた文字列リテラルに、マルチバイト文字を含めた場合、必要な配列要素数（バイト数）がいくつになるかは、文字エンコードによって変化するので、「`char a[]="あいうえお";`」というような場面で、要素数を省略するのは理にかなってます。

9.2.2　文字列のコピー

strcpy() は、文字列をコピーする標準ライブラリ関数です。やはり動作を理解するために、自作しなおしてみたのがソースコード 9.2 です。

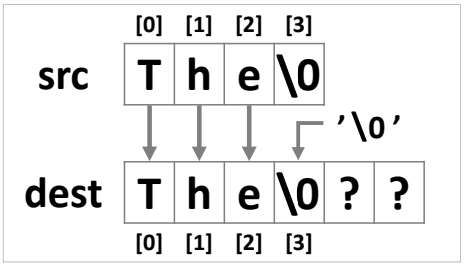
str_copy(dest, src)　ソースコード 9.1 の str_length() と同じ for ループで、1 文字ずつ src[] から dest[] にコピーします。src は source（源泉、情報源）、dest は destination（目的地）の省略形で、それぞれコピー元とコピー先を表す変数名としてよく使われます。

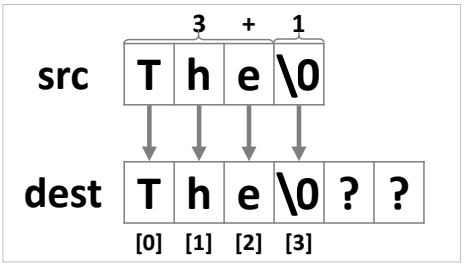
ループ終了後に、dest[] の末尾に終端のヌル文字を書き込みます。ループ変数 i は、最後の dest[i] への代入後にも 1 増えているので（7 行目）、ループ後のヌル文字の代入（10 行目）で i に +1 のような調整項が不要と、うまくできています。

ヌル文字忘れは、このあとのコラムで説明するように、大事故につながります。

ソースコード 9.2　文字列をコピー

```
1  #include <stdio.h>
2  #include <string.h> // strlen(), strcpy()
3
4  /* strcpy()の自作関数：文字列をコピーする */
5  void str_copy(char dest[], char src[]) {
6      int i; // ループ後にも必要
7      for (i=0; src[i]!='\0'; i++) {
8          dest[i] = src[i];
9      }
10     dest[i] = '\0'; // [i+1] ではない
11 }
12
13 /* strcpy()の自作関数：文字列をコピーする（その２） */
14 void str_copy2(char dest[], char src[]) {
15     int len = strlen(src);
16     for (int i=0; i<len+1; i++) { // +1 は '\0' のため
17         dest[i] = src[i];
18     }
19 }
20
21 int main(void) {
22     char buff[100];
23     strcpy(buff, "abcdef");  printf("%s\n", buff); // "abcdef"
24     str_copy(buff, "ABCD");  printf("%s\n", buff); // "ABCD"
25     str_copy2(buff, "123");  puts(buff);           // "123"
26     return 0;              // puts() は文字列用の表示関数
27 }
```





str_copy2(dest, src) 同じようにコピーする関数です。こちらでは、あらかじめ strlen() で src[] の長さを調べておいて、回数の決まったループを使いました。この手段のほうが安全ですし、理解しやすいかもしれません。ただし、src[] を全体として 2 回走査するので、少し効率が落ちます。

終端のヌル文字は、ループ後に dest[len]='\0' と代入してもよいのですが、ループ回数を文字数よりも 1 回多めにしても同じ効果があります。このとき、0 始まりの**良いループ**（☞**??**節）になるよう、条件は「i<=len」よりも、イコールのない「i<len+1」がよいでしょう。そして「+1」の理由をコメントに書いておきます（16 行目）。

コラム： 良いコメント

コメントには、後から読む人の役に立つことを書きましょう。ちなみに、**明日の自分は他人**なので、他人のために書いたコメントは、後から自分の役に立ちます。

コメントには、表面的な動作を説明するのではなく、このような処理をしている**理由を説明する**のが良いと言われています。例えば、ソースコード 9.2 の 16 行目のように、標準から外れた動作をしている理由を書き留めておくとよいでしょう。

コラム： ヌル文字忘れ

文字列の終端に'\0' を付け忘れると、恐ろしいことが起こります。

文字列 str は printf("%s", str); あるいは puts(str); で表示します。この str の終端に'\0' がなかったとしたら、表示する内容は、str の末尾から続くメモリに突入し、そのまま'\0' が出てくるまで進みます。別のプログラムに割り当てられている領域にまで侵入すると、OS のメモリ保護機能の働きによっては（本当に表示する前に）プログラムが強制停止されます。

このように、'\0' を忘れるだけで**文字列表示さえできなく**なることがあります。

強制停止させられると、プログラムの間違いに気づけるので、むしろ幸運かもしれません。また、このような不正アクセスを積極的に検出するツールも存在します。

9.2.3　文字列の連結

strcat() は、文字列の後ろに別の文字列を連結する標準ライブラリ関数です。cat は、concatenate（連結する）の大胆な省略形です。ソースコード 9.3 に自作しなおしました。

str_concat(dest, src)　ソースコード 9.2 の str_copy() と同じ for ループで、1 文字ずつ src[] から dest[] にコピーします。ここでは、元々 dest[] に格納されていた文字列の後ろにコピーするので、元の文字列の長さを len に代入して、len だけずらしてコピーします。dest[] の末尾に終端のヌル文字を書き込む必要があるのも str_copy() と同じです。

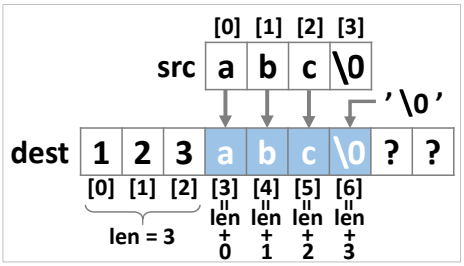
str_concat2(dest, src)　同じ動作ですが、ポインタの知識があれば、コピー先の位置をずらして strcpy() を呼び出すだけですみます。9.3 節を学んでから見直してください。

main()　strcat() の使い方にはコツがあります。まず大きめの作業領域を用意します。ここでは 100 バイト（99 文字分）の buff[] を用意して、空文字列で初期化しました。そして、strcat() を繰り返して、この後ろに別の文字列を連結していきます。

ただし、格納先の buff[] があふれないことを保証せねばならないので、実用的には、strcat() の前に strlen(dest)+strlen(src)+1 <= sizeof(dest) のような検査が必要です。

ソースコード 9.3　文字列を連結

```
1  #include <stdio.h>
2  #include <string.h> // strlen(), strcpy(), strcat()
3
4  /* strcat()の自作関数：文字列を連結する */
5  void str_concat(char dest[], char src[]) {
6      int i, len = strlen(dest);
7      for (i=0; src[i]!='\0'; i++) {
8          dest[len + i] = src[i]; // len だけずらしてコピー
9      }
10     dest[len + i] = '\0';
11 }
12
13 /* strcat()の自作関数：文字列を連結する */
14 void str_concat2(char dest[], char src[]) {
15     int len = strlen(dest);
16     strcpy(dest + len, src);// ポインタ演算でコピー先アドレスをずらす
17 }
18
19 int main(void) {                // 100 は適当な大きめの値
20     char buff[100] = "";         // buff : ""
21     strcat(buff, "123");         // buff : "123"
22     str_concat(buff, "abc");     // buff : "123abc"
23     str_concat2(buff, "XYZ"); // buff : "123abcXYZ"
24     puts(buff);
25 }
```



コラム： 空文字列で初期化

int の変数を 0 で初期化するように、char 配列を長さ 0 の空文字列で初期化したいことはよくあります。方法はいくつも考えられます。

```
/* 初期化（変数定義と同時） */

char s[100] = "";      // 直感的
char t[100] = {'\0'}; // 等価
```

```
/* 変数定義と分離した代入 */
char s[100], t[100];
strcpy(s, ""); // 文字列コピー
t[0] = '\0';   // 簡便
```

ダブルコーテーション (") を 2 個連続すれば、長さ 0 の空文字列リテラルです。これを初期化子のブロックの代わりにするのが**直感的**です。変数定義と分離するなら、strcpy() で**空文字列をコピー**するのがとりあえずの作法です。関数呼び出しが大げさだと思えば、先頭要素にヌル文字を代入するのも**簡便**でよいでしょう。

ソースコード 9.3 では buff[] を空文字列で初期化していますが、最初の文字列の "123"だけ strcpy() でコピーすれば、初期化の必要がなくなります。

コラム： 文字列の実現方法

C 言語の採用した、ヌル文字で文字列の終端を表すという方法は、文字列を実現する唯一の方法というわけではありません。

長さと、文字列そのものを別に管理するという方法もあります。これなら文字列にどんな文字でも含められますが、逆に理論上は長さに上限ができます。

C++ の std::string、Perl や Python の文字列もこの方法で実現されているようです。長さを 32 ビット整数で管理していれば数 GB 分格納できるので、事実上制限なしといえるでしょう。

9.2.4　書式変換

sprintf() は、文字列用の関数というわけではなく、標準入出力のための <stdio.h> で宣言されています。いつもの printf() は画面（コンソール）に出力しますが、sprintf() は画面の代わりに文字列に書き込みます。接頭辞の s は string の意味です。

snprintf() は sprintf() の変形で、書き込まれる文字列のバイト数を制限します。あふれた部分は書き込みません。最後に必ずヌル文字が付きます。

これらの書式変換の機能は、文字列操作にも役立ちます。まずは文字列連結です。

```
char s1[] = "123";
char s2[] = "abc";
char buff[100];

/* 文字列連結 */
sprintf(buff, "%s%s", s1, s2);                  // buff : "123abc"
snprintf(buff, sizeof(buff), "%s%s", s1, s2); // buff : "123abc"
```

次は、atoi() の反対の動作、つまり数値を文字列に展開に利用してみます。

```
/* 文字列→数値 */
int    i = atoi(buff);
double f = atof(buff);
```

```
/* 数値→文字列 */
snprintf(buff, SIZE, "%d", i);
snprintf(buff, SIZE, "%f", f);
```

snprintf() は C99 で導入された[*3]新しい関数ですが、単純な記述で**バッファオーバーフロー** (buffer overflow) （不正な配列操作などによるメモリ破壊）を防げますので、活用したいものです。strcpy() などのほうが実行効率はよいでしょうが、少し油断すると、すぐにヌル文字がつかなかったり、配列サイズをあふれてしまいます[*4]。

コラム：金額の 3 桁ごとのコンマ

大きな金額を表記する際に、3 桁ごとにコンマを入れて「123,456,789 円」などとすることがあります。ところが C 言語には、この表記を実現する直接の機能はありません。このような表記は、国や文化によって異なるため、サポートされなかったものと思われます。

ちなみに、小数点を表すのに、我々はピリオド（.）を使いますが、これにコンマ（,）を使う国もあります。

[*3] C99 で導入される前から、POSIX という Unix 系の規格に存在していて、一部では使われていました。

[*4] 確かに snprintf() はバッファオーバーフローを防げますが、バッファが足りないまま処理を続けるとおかしな動作になります。したがって、バッファ不足を検出してエラー処理せねばなりませんが、POSIX 時代と C99 とで snprintf() の戻り値が微妙に異なるため、環境依存してしまいがちなのが惜しいところです。

鋭意作成中

コラム： サイズ上限付き文字列

<stdio.h> に**新しく**加わった snprintf() は、書き込む文字列領域のサイズを指定できて、その文字列の終端には**必ずヌル文字**を書き込みます。

<string.h> に**古くから**ある関数の中にも、strncpy() や strncat() のように、領域のサイズを指定できるものがあります。しかしどうやらこれらは、終端を表すのにヌル文字だけに頼らず、**サイズ上限も終端とする**、少し変わった文字列のデータ構造[a]を前提としているようです。つまり、領域いっぱいの文字列を作ると、終端の**ヌル文字が抜け落ちる**関数が混ざっています。

このデータ構造は、かつてよく使われた、固定長領域のデータを扱うのには役に立つのでしょうが、画面表示には puts() が使えず、printf() で文字数の最大値を指定する必要があって、扱いが難しいので、我々は利用しないほうがよいでしょう。

初期化子の文法にもこのデータ構造への用意があって、「`char s[3] = "123";`」のように、要素数がヌル文字分を含めなくても警告されないので、要注意です。

なお、C11 で「境界チェックインタフェース」として追加された strcpy_s() など xxx_s() の関数群は、バッファオーバーフローを防ぐ工夫がされているものの、機械的な置き換えには難があったり、立場による思惑の違いからか、可搬性に問題のある状態が続いています。

[a] 文字列を部分的に書き換えることも意図されているようです。

9.2.5　文字列の比較

strcmp() は、文字列を**辞書順** (dictionary order) で大小比較する標準ライブラリ関数です。cmp は compare（比較する）の、よくある省略形です。

辞書順というのは、文字通り辞書に現れるような順序です。アルゴリズム的に記述すると、次のようになります。まず先頭の文字を比較します。異なる文字であれば、この文字の文字コードで大小が決まります。同じ文字なら、次の文字を比較します。次の文字も同じなら、さらにその次と、繰り返します。同じ文字が続いてどちらかの文字列の末尾に達したら、文字列の長い方を大きいと判断します。同じ長さなら、等しいと判断します。

strcmp(s1, s2) の戻り値は、s1 が s2 に比べて、大きければ正の値、小さければ負の値、等しければ 0 です。文字列に限らず、**比較関数の戻り値が「プラス、マイナス、0 のどれか」**というのは、よくある取り決めです。+1、−1 と決めていないのは、プラスとマイナスをまとめて return a-b; のように処理できる場合があるからです。

```
char s1[] = "hoge";
char s2[] = "fuga";
if (s1 < s2) { ... }            // × この表記ではアドレスの比較になる
if (strcmp(s1, s2) < 0) { ... } // ○ 0と比較する不等号は、上と同じ
```

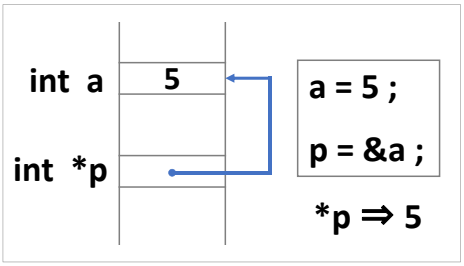
この取り決めに従うと、呼び出し側では、上の例のように「s1 と s2」の間で使いたい比較演算子を、「関数値と 0」の間で使えばよいことになります。上記は < の例ですが、== でも != でも同じ方法で大丈夫です。使用例は 18 ページのソースコード 9.8 を参照してください。

ソースコード 9.4 に strcmp() の実装例[*5]を載せておきます。4 行目で unsigned char にキャストしているのは、言語規格でこのように比較するよう指示があるからです。この比較が、異なる文字だけでなく、短い文字列の末尾（つまりヌル文字）を小さいと判断するところまで、うまく処理してくれます。

ソースコード 9.4　文字列の比較

```
1 /* strcmp()の自作関数：文字列を比較する */
2 int str_compare(char s1[], char s2[]) {
3     for (int i=0; s1[i]!='\0' || s2[i]!='\0'; i++) {
4         int diff = (unsigned char)s1[i] - (unsigned char)s2[i];
5         if (diff != 0) { return diff; } // 異なる
6     }
7     return 0; // 等しい
8 }
```

[*5] これを見てわかるように、1 バイトずつを文字コードで比較しているので、マルチバイト文字が混じると、同一判定は大丈夫ですが、大小判定はおかしくなる可能性があります。標準ライブラリ関数の strcmp() でも事情は同じです。



9.3 ポインタ

通常は、変数に割り当てられたメモリ領域を「変数名」で区別します。領域はコンパイラによって、必要なときに自動的に割り当てられ、不要になれば開放されます。割り当てられた**メモリ領域が途中で変わることはありません**。これは配列でも同じです。領域がいつ、どのように割り当てられるかは、プログラムで気にする必要は（ほぼ）ありません。

ポインタ (pointer) 変数を使うと、メモリ領域の割り当てをプログラムで管理できます。メモリ領域を、CPU が管理する上で用いる「アドレス（番地）」で区別します。ポインタ変数の値（=アドレス）を変更すると、ポインタの指し示す**メモリ領域が変わります**。このおかげで、柔軟な処理ができる反面、指し示す領域に何が保管されているかを、プログラマが意識して管理する必要があります。

ポインタにも「**型**」があります。アドレスの指し示すメモリ領域にある変数の型です。どの型のポインタであるのかは、しっかりと認識せねばなりません。変数の定義では、int へのポインタであれば「`int *p;`」と、* を付けます。複数のポインタ変数をまとめるときは、「`int *q, *r;`」と、それぞれに * が必要です。（このおかげで、通常の変数の定義と混在できます。）

```
int a;      // intの普通の変数
int b, c;   // まとめて定義
```

```
int *p;     // intのポインタ変数
int *q, *r; // まとめて定義
int d, e, *s, *t; // 混在
```

このように、ポインタは、領域を「変数名」ではなく「数値（番地）」で区別する手段です。この点で、ポインタと配列が似ていると思った人もいるかもしれません。そのとおり、**ポインタと配列は表裏一体**です。9.2 節で char s[] と char *s が同じだと説明したように、互いに交換できる場面まであります。

表 9.1　ポインタの演算子

演算子	演算の種類
&	アドレス
*	間接参照

表 9.2　通常の変数とポインタ変数の関係

	変数定義	アドレスの参照	int の参照
通常	`int x;`	`&x`	`x`
ポインタ	`int *p;`	`p`	`*p`

9.3.1　ポインタを扱う演算子

表 9.1 にまとめたように、変数のアドレスを取り出すのが**アドレス** (address) 演算子 **&** です。逆に、アドレスの先にある変数の値を取り出すのが**間接参照** (indirection) 演算子 ***** です。この * は、先ほどの変数定義の * とは役割が違って**作用はほぼ正反対**ですが、次のように、混同すると（かえって）**つじつまが合う**ように巧妙に作られています。

「`int *p, x;`」と定義すると、`p = &x;`（アドレスの代入）と `*p = x;`（int の代入）が両辺の型の一致する操作です（表 9.2）。

「`int *p;`」は、もともと「**p は int ポインタ 型**」の意味ですが、（* を間接参照かのように）「*p」を変数名と考えると、「***p は int 型**」とも解釈できます。

ただし、**初期化**の場面での * には注意してください。変数への代入を短縮表記しているだけですから、* は変数定義の一部として解釈する必要があります。

```
/* ポインタ定義の2通りの解釈 */
int *p;  // 「p は int*」
         // 「*p は int」
int x = 5;
p = &x;  // アドレスの代入
*p = x;  // intの代入
```

```
/* 初期化 */
int x = 5;
int *p = &x; // アドレスで初期化
```

9.3.2　配列とポインタの関係

`int a[10];` の a は、ポインタとしての機能もあるので、`a[0]` と `*a` が同じ意味です。`a[1]` は `*(a+1)` が同じです。* の優先順位が高いので、アドレスの加算を先にするよう、このカッコが必要です。`int *p = a;` とポインタ変数を用意すると、`p[0]` や `*p` が `a[0]` と同じです。`int *q = a+1;` と先にアドレスを加算しておくと、`q[0]` と `a[1]` が同じです。a は配列のため、アドレスは変更できませんが、ポインタ変数は `p++;` や `q += 4;` のようにして、アドレスを変更できます。下の表の水色部分は、よく使う表現です。

変数定義（初期化）	`int a[10];`		`int *p = a;`		`int *q = a+1;`	
a の 0 番目要素	`a[0]`	`*a`	`p[0]`	`*p`	`q[-1]`	`*(q-1)`
a の 5 番目要素	`a[5]`	`*(a+5)`	`p[5]`	`*(p+5)`	`q[4]`	`*(q+4)`
アドレスの加減算	**不可能**		`p++;`		`q += 4;`	
a の 5 番目要素	`a[5]`	`*(a+5)`	`p[4]`	`*(p+4)`	`q[0]`	`*q`

9.3.3 関数に渡すポインタ

関数の引数をポインタにすると**参照渡し**[*6]になります（☞**??**項）。**??**ページのソースコード**??**で失敗したスワップ関数が、ソースコード 9.5 のように実現できます。呼び出す側では、実引数の変数に & を付けます。関数内では、*x と *y を int 変数と思って入れ替えます。これで main() 側の a と b が（main() では代入していないのに）入れ替わります。（値の変わるはずのない）定数に & を付けると、コンパイルエラーになります。

ソースコード 9.5 スワップ関数

```
1  #include <stdio.h>
2
3  void swap(int *x, int *y) {          //  x と  y はintへのポインタ
4      int tmp = *x; *x = *y; *y = tmp; // *x と *y はint
5  }
6
7  int main(void) {
8      int a = 1, b = 2;
9      swap(&a, &b); // & でポインタにする（アドレスを取り出す）
10     printf("main: a=%d, b=%d\n", a, b); // a=2, b=1
11 //  swap(&1, &2); // 定数には & は付けられない（コンパイルエラー）
12 }
```

scanf() が変数の値を書き換えることができるのも、そして引数の変数を & 付きで渡す必要があるのも、同じ仕組みです。

配列をコピーする関数は、**??**ページのソースコード**??**で array_copy() を作りましたが、ソースコード 9.6 のように、ポインタとしてもコピーできます。仮引数では、int dst[] は int *dst と解釈されるので、array_copy() と pointer_copy() の役割はまったく同じです。

ソースコード 9.6 配列のコピー（ポインタ版）

```
1  void pointer_copy(int n, int *dst,  int *src) {
2      //           (int n, int dst[], int src[]) でも同じ
3      for (int i=0; i<n; i++) {
4          *dst++ = *src++;                // ポインタでコピー（短縮）
5       // *dst   = *src;  dst++;  src++;  // ポインタでコピー（分解）
6       // dst[i] = src[i];                // 配列としてコピーもOK
7      }
8  }
```

*dst++ = *src++; の表記は、*dst = *src; とアドレスのインクリメントをまとめたもので、CPU にとっても効率のよい動作に近いのですが、昨今の最適化技術の進んだコンパイラであれば、dst[i] = src[i]; も含め、どの表記でも効率に大差はないでしょう。

[*6] ポインタ変数としては、（相変わらず）値渡しですが、ポインタの指すもので考えると参照渡しです。

9.4 文字列とポインタ

文字列を扱うのに、char 配列にするか、char のポインタにするかで、2 通りの手法があります。ソースコード上の "The" は、char a[4] = "The"; なら**配列の初期化子**ですし、char *p = "The"; なら**文字列リテラルへのポインタ**と、役割が違います。

```
/* char配列 */
char a[4] = "The"; // 初期化子
char b[4] = {'T','h','e','\0'};
a[0] = 's'; // ○ 1文字の書き換え
//a = "the";// NG アドレス変更不可
//a++;      // NG アドレス変更不可
printf("%zu", sizeof(a)); // 4
```

```
/* charポインタ */
char *p = "The"; // 文字列リテラル
//char *q={'T','h','e','\0'}; //NG
p[0] = 's';  // × 動作保証なし
p = "the";   // OK ポインタの代入
p++;         // OK "he"
printf("%zu", sizeof(p)); // 8
      // (64 bitアドレスの場合)
```

char a[4]; では、a のメモリ割付が固定されるので、a = "the"; とか a++; のような**アドレスの変更ができません**。一方、char *p; では、文字列リテラルを指すポインタなので、p = "the"; とか p++; のような**アドレスの変更が可能です**。ただし、**文字列リテラルは ROM 領域**に割り当てられる可能性があるため、p[0] = 's'; のような 1 文字単位の書き換えの動作保証がありません。（しかも、実行時エラーになるとも限りません。）

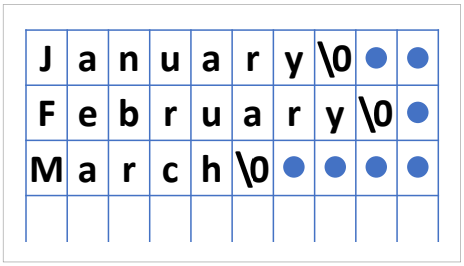
sizeof による大きさは、char 配列では (文字数)+1 です。+1 はヌル文字分です。char ポインタなら、文字数には影響されず、ポインタ変数に共通する大きさ（典型的には 32 bit アドレスなら 4、64 bit アドレスなら 8）の固定値です。

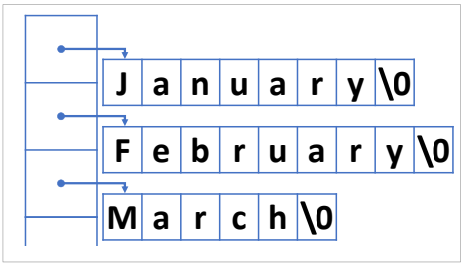
コラム：文字列リテラルの共有

「char *p = "The", *q = "The";」とした場合、コンパイラによっては、p と q のアドレスが同じになる、つまり同一の文字列リテラルを共有することがあります。C 言語規格では、**文字列リテラルは ROM 領域**に割り当てられることを想定しているので、共有を許す代わりに、逆に書き換えが保証されません。

9.4.1 文字列の配列

いくつもの文字列をまとめて扱うには、文字列の**配列**を利用したくなります。C 言語では 2 通りの実現方法があります。どちらを採用するかは (1) メモリに無駄がないか (2) プログラムの実行中に変更するか (3) 関数に渡しやすいか、などで判断します。現実には (3) を重要視して、これから紹介する**後者**を採用することが多いでしょう。ソースコード 9.7 を見ていきます。





char の 2 次元配列 month_name は char の 2 次元配列です。12 ヶ月分、一番長い月名に合わせて 10 バイトずつの領域を割り当てました。短い名前の後ろには使わない無駄なメモリができますが、この長さを上限として、実行中に変更できます。

char のポインタの配列 month_name2 は char* の配列です。初期化子のブロックは先ほどとまったく同じです。ポインタの指す先には、各月名の長さちょうどの文字列リテラルが用意されます。メモリに無駄はありませんが、ポインタ配列の領域が余分に必要なので、長さのばらつきが少ないと、節約にならないこともあります。そして、ポインタの先の文字列リテラルは定数なので、(1 文字単位の) 代入の動作が保証されません。(ポインタの変更、つまり文字列の差し替えは可能です。)

16 行目のような場面では、使い方に区別はありません。C 言語の巧妙なところです。

ソースコード 9.7 月名の配列

```
1  #include <stdio.h>
2  #define N_MONTH 12
3
4  char month_name[N_MONTH][10] = { // charの2次元配列
5      "January", "February", "March", "April", "May", "June",
6      "July", "August", "September", "October", "November", "December"
7  };
8
9  char *month_name2[N_MONTH] = {   // charポインタの配列
10     "January", "February", "March", "April", "May", "June",
11     "July", "August", "September", "October", "November", "December"
12 };
13
14 int main(void) {
15     for (int i=0; i<N_MONTH; i++) {
16         printf("%s %s\n", month_name[i], month_name2[i]);
17     }
18 }
```

ただし、関数に渡すことを考えると、1 次元配列ですむ、ポインタの配列が有利です。ソースコード 9.8 では、辞書順で比較した最小の文字列を返す `str_min(n, s)` を作りました。文字列の配列は `char *s[]` と、ポインタの配列で受け取ります。もしも 2 次元配列で受け取るなら、文字列のサイズごと（例えば `char s[][10]` と `char s[][11]`）に、別の関数を作らねばなりません。

ソースコード 9.8　辞書順で最小の文字列

```c
#include <stdio.h>
#include <string.h> // strcmp()
#define N_MONTH 12

char *month_name2[N_MONTH] = {   // charポインタの配列
    "January", "February", "March", "April", "May", "June",
    "July", "August", "September", "October", "November", "December"
};

char *str_min(int n, char *s[]) {
    char *min = s[0];
    for (int i=1; i<n; i++) { // 先頭要素を除外したループ
        if (strcmp(min, s[i]) > 0) { min = s[i]; }
    }
    return min;
}

int main(void) {
    printf("%s\n", str_min(N_MONTH, month_name2)); // April
}
```

頻出ミス

C 言語のプログラムを作っていて、文字列を扱うのに strcpy() や strcat() を駆使して、snprintf() も正しく使えるようになって、やれやれと思っている人がいるかもしれません。

そのような人には申し訳ないですが、そもそも**文字列操作を主体**とするプログラムは、**C 言語ではやめておく**のが賢明です。メモリの不正アクセス検出機能がなくて、間違いに気づけないので**練習にもなりません**。世間を騒がすセキュリティホールの原因の多くが、ヌル文字の付け忘れや、領域サイズの計算間違いによるバッファオーバーフローであることを肝に銘じておきましょう。

文字列操作をするなら、文字列を単なるバイト列ではなく、オブジェクトとして操作する Python のような**スクリプト言語で開発することを強くおすすめ**します。これは、くれぐれもです。

鋭意作成中

コラム： 複合リテラル

関数の実引数に配列を渡す場合、char 配列なら、変数に代入することなく、文字列リテラルを直接渡せます。それでは、char 以外の配列はどうでしょうか。

```
void func(char x[]);

int main(void) {
    func("string");
    ...
```

```
void func2(int x[]);

int main(void) {
    func2( (int[]){1,2,3} );
    ...
```

C99 から、このような変数と結びつかない配列である**複合リテラル** (compound literal) が使えるようになりました。ただし C++ ではサポートされてないからか、スコープが該当ブロックに限られていているからか、それとも要素数を計算できないからか、それほど使われている場面を見かけません。反面、Java での類似の機能「無名配列」はよく使われています。

9.5 練習問題

1. **[可搬性（☞9.2 節、9.2.1 項）]**
   会社員 A「strlen() の値を printf() で表示したくて、試行錯誤の末に"%u"の書式文字列なら警告されないことを突き止めた。そして以下のプログラムを作成して、手元で正しく動作することも確かめた。」

   ```
   printf("文字列の長さは%u\n", strlen("abc")); // NG
   ```

   しかし会社はこのプログラムを採用しない。その理由を説明せよ。
2. **[文字列操作（☞9.2.1 項）]**
   4 ページのソースコード 9.1 を参考に、**??**項の isdigit() を用いて、次の関数を作れ。
   - `int str_ndigit(char str[])` は、str[] に含まれる数字の個数を返す。

3. **[文字列操作（☞9.2.2 項）]**
   6 ページのソースコード 9.2 を参考に、次の関数を作れ。
   - `void str_copy_toupper(char dest[], char src[])` は、src[] から dest[] へ、英小文字を大文字に変換しながらコピーする。
     ヒント：**??**項の `toupper()` は、小文字を大文字に変換し、それ以外の文字は変換せずにそのまま返す。
   - `void str_copy_swap_case(char dest[], char src[])` は、src[] から dest[] へ、英字の大文字と小文字を入れ替えながらコピーする。
     ヒント：**1 文字の**大文字と小文字を入れ替える関数を作り、繰り返し呼び出せ。

4. **[文字列比較（☞9.2.5 項、9.4.1 項）]**
   気象庁の発表する、6 種類の特別警報と 7 種類の警報について、所属機関が休講になるかどうかを表示せよ。
   警報の種類や、所属機関の規定は、Web 等を活用して調べよ[*8]。

   【実行例】
   - 「暴風雪警報」→ 休講
   - 「大雪警報」→ 開講
   - 「特別警報（大雪）」[*7]→ 不明

5. **[文字列操作（☞??項）]**
   スパイから君に、右の暗号文が届いた。スパイは、文字列の前後を入れ替え、文字の ASCII コードを +3 ずらして暗号文を作っているらしい。逆の操作で、原文に戻してくれ。

```
(暗号文) xndnxrj0xndqrpdp
```

```
(原文)123abc → (暗号文)fed654
```

6. **[文字列操作]**
   文字列で表された数値を int に変換する処理を自作せよ。
   ヒント：文字列の先頭から 1 文字ずつ調べ、ヌル文字が出てくるまで、「これまでの数を 10 倍して今の数を加える」を繰り返せばよい。下の char2int() も参照せよ。

   "1"　⇒　0 * 10 + 1 = 1
   "12"　⇒　(0 * 10 + 1) * 10 + 2 = 12
   "123"　⇒　((0 * 10 + 1) * 10 + 2) * 10 + 3 = 123

```
int char2int(char c) { return c - '0'; } //   1文字の数値変換
```

7. **[ループ処理への書き換え]** ☞**??**項
   TODO: ポインタの課題:slice() 処理で*dest++=...

[*7] このような警報はありません。気象庁の用語では「大雪特別警報」です。
[*8] 休講の条件のみが記載されていると、記載のないものが本当にすべて**開講**なのか、あるいは検討されてなくて**不明**なのか、大いに迷うところです。プログラムの仕様書なら、休講と開講の両方の条件を列挙して、**それ以外を不明**とすべきでしょう。（あるいは、事前に警報の種類をコード化しておきます。）

索引