第 7 章

関数（2）

本書では、一般的な入門書とは異なり、関数の作り方を早い段階で紹介しました。そして、手間のかかる約束事の紹介を後回しにしました。これは、言語にかかわらず通用する、**大きく複雑な作業を、小さな関数に分割して単純化して解決する**という体験を、プログラミングの早い段階で経験してもらいたかったからです。

さてここでは、後回しにしていた約束事を紹介します。C 言語に特有のことも多いため、難易度は高く感じるかもしれませんが、少しずつ身につけていけば大丈夫ですので、安心してください。

キーワード

- プロトタイプ宣言
- 暗黙的な宣言
- 標準ライブラリ関数のヘッダ
- スコープ・寿命
- グローバル・ローカル・static
- スネークケース・キャメルケース

7.1　プロトタイプ宣言

これまでのプログラムでは、ソースコード上での位置でいうと、呼び出される関数を先に記述して、後から呼び出していました。そして、最初に実行される main 関数を、ソースコードの最後に書いていました。この**順序制約**は、関数の間の呼び出し関係が明らかならば気にするほどでもないでしょうが、長いプログラムになってきたり、2 つの関数がお互いに呼び出しあっているなど、逃れたくなる場面もあります。

ソースコード 7.1　プロトタイプ宣言なし

```
1  #include <stdio.h>
2
3  // プロトタイプ宣言なし
4
5  int main(void) {
6      printf("正方形の面積は%fです。\n", square(1.0));
7      return 0;
8  }
9
10 double square(double x) { return x * x; } // 関数定義
```

例を見てみましょう。ソースコード 7.1 は順序制約を破っているので、コンパイル時に次のような警告とエラーが出ます。

ソースコード 7.1 のコンパイル出力（gcc の場合）

```
square.c: 関数 ‘main’ 内:                          ← 6 行目 5 文字目の意味
square.c:6:5: 警告: 関数 ‘square’ の暗黙的な宣言です
     printf("正方形の面積は%fです。\n", square(1.0));
     ^                                              ← 5 文字目の位置の目印
                                                    （1 つの警告メッセージ）

square.c: トップレベル:                            ← 10 行目 8 文字目の意味
square.c:10:8: エラー: ‘square’ と型が競合しています
 double square(double x) { return x * x; }
        ^                                           ← 8 文字目の位置の目印
square.c:6:50: 備考: 前の ‘square’ の暗黙的な宣言はここです
     printf("正方形の面積は%fです。\n", square(1.0));
                                                ^
                                                競合相手の場所
                                                    （1 つのエラーメッセージ）
```

このメッセージは、意味がわかりにくいのですが、6 行目で square() を呼び出すまでに、この関数の戻り値や引数の型が判明していない、ということを**間接的**に表現しています。（メッセージの意味は 7.1.1 項で読み解きます。）

関数の、引数などを含めた型のことを**関数プロトタイプ** (function prototype) といいます。コンパイラは、関数呼び出しが正当であるかを検査して、例えば引数の個数が違っていればコンパイルエラーにしてくれますが、この検査に関数プロトタイプが必要です。コ

ンパイラは、ソースコードを上から順番に解釈してゆきますから、後に記述してあることを知らなくてもよいように、ソースコード上の記述順序に制約[*1]が設けられています。

この順序制約を実質的に取り払う方法があります。後から出てくる関数の「予告」を事前に書けばよいのです。予告のことを関数の**プロトタイプ宣言** (prototype declaration) といいます。ソースコード 7.2 では、3 行目を書き加えたことで、エラーや警告が解消されました。3 行目のプロトタイプ宣言は 10 行目の関数定義とほとんど同じ、違いは関数本体のブロックを**セミコロンで置き換え**てあることだけです。

ソースコード 7.2 プロトタイプ宣言あり

```
1  #include <stdio.h>
2
3  double square(double x); // プロトタイプ宣言
4
5  int main(void) {
6      printf("正方形の面積は%fです。\n", square(1.0));
7      return 0;
8  }
9
10 double square(double x) { return x * x; } // 関数定義
```

自分のプログラムでプロトタイプ宣言を書くときは、**手でタイプしなおすと間違える**ので、テキストエディタで完成している**関数の先頭部分をコピー**します。そして、関数ブロックを消して、代わりにセミコロンで置き換えます。プロトタイプ宣言を書く場所は、関数呼び出しよりも前であればどこでもよいのですが、中途半端なことはせず、**#include のすぐ下**の行に書くのが習慣です。

なお、関数定義（本体）の現れた後は、正しく関数呼び出しができますから、関数定義もプロトタイプ宣言の役目を含んでいることがわかります。

コラム： プロトタイプ宣言は無駄?

プロトタイプ宣言を「同じことを 2 度も記述して無駄」とか「自動生成できてしかるべき」と思う人は、プログラミングの才能があります。現実の作業として、プログラムを作っているうちに引数が足りないことに気づいて後から追加する、などという場面は頻繁にあって、そのたびにプロトタイプ宣言まで修正する（コピーしなおす）のは煩雑かつミスが入りやすいです。ですから、関数の記述順序を調節して、**プロトタイプ宣言を書かずにすませる**という流儀もよく見かけます。

[*1] この制約は、C 言語が登場した頃のコンピュータの能力からくるもので、今の進歩したコンピュータの能力や高性能なコンパイラ技術から考えると、コンパイラを甘やかせ過ぎているようにも感じられます。実際、Java や C#のような新しい言語には記述順序に制約はありません。

ただし、分割コンパイル（1 つのプログラムを複数のソースコードで構成する）を行うときには、ファイルを越えた呼び出しを許す（つまり公開する）関数だけを選び出して、あるファイル[a]にプロトタイプ宣言を書き並べる必要があります。この選び出す作業の存在が、自動生成を難しくしています。

[a] あるファイル（＝共有するヘッダファイル）で関数のスコープを制御するという仕組みは、単純かつ効果的ではありますが、時代を感じずにはいられません。ちなみに Java 言語では、プロトタイプ宣言もヘッダファイルもなく、メソッド（≈ 関数）に public という修飾子をつけることで、クラス（≈ ファイル）を越えた呼び出しを許したことになります。

7.1.1　暗黙的な宣言

ここで紹介するのは C99 で廃止されたはずの機能ですが、一部のコンパイラになぜかまだ残っています。また、廃止されたコンパイラでのエラーメッセージを理解するのにも役立つ知識です。

プロトタイプ宣言を忘れると何が起こるのでしょうか。2 ページのソースコード 7.1 のコンパイル出力をもう一度見てみてみましょう。関数を呼び出そうとした時点ではまだエラーにならず、**暗黙的な宣言** (implicit declaration) を行ったという警告[*2]になっています（6 行目）。これまでに明示的なプロトタイプ宣言がなかったので、事前に決められたプロトタイプで代用するというのがこの機能です。**事前に決められたプロトタイプ**とは `int func();` という、関数は int 型を返し、引数は（ないわけではなく）**型検査をしない恐ろしいもの**[*3]です。

構文解析が進んで、関数本体のプロトタイプが、初めの暗黙的な宣言と矛盾したので、めでたくエラーとして検出されました（10 行目）。この長いメッセージで、矛盾する宣言がどこにあったのかまで教えてくれています。square() が double 型を返すので、暗黙的な宣言の int 型と食い違ったということです。

しかし、運悪くプロトタイプが矛盾しなければ、この段階は警告にもなりませんし、全体を通じてエラーが起こりません。運悪くといっても、関数が int 型でさえあれば**引数は検査の対象外**ですから、簡単に起こることです。ですから、**最初の警告**を見逃さないようにすることが重要です。

[*2] 暗黙的な宣言は、C 言語の初期の規格 (K&R) で書かれたソースコードをエラーにしないために、言語規格の 2 世代め (C89) で考案された巧妙な仕組みです。このため「プロトタイプ宣言がない」との直接的なメッセージになりません。C99 では廃止されたのですが、それでも「暗黙的な宣言は無効」のような、わかりにくいエラーメッセージになることがあります。C23 からはプロトタイプ宣言が必須になったので、直接的なメッセージでエラーになるはずです。

[*3] C23 以前では、プロトタイプ宣言の引数は、`()` と空にすると型検査の対象外になります。引数なしを示すには、いつもの main 関数のように、`(void)` と書く必要があります。

7.1.2 標準ライブラリ関数のプロトタイプ宣言

C 言語で用意されている標準ライブラリ関数であっても、呼び出す前にはプロトタイプ宣言が必要です。このようなプロトタイプ宣言は、システムの用意する**ヘッダファイル** (header file) に記述されているので、#include 命令[*4]で読み込みます。printf() のプロトタイプ宣言は <stdio.h> というファイルに書かれているので、これまでのプログラムの先頭には「#include <stdio.h>」を書いてきました。数学関数を使う場面では、「#include <math.h>」も追加しましたが、このような仕組みになっていたのです。

コラム： 標準ライブラリ関数の調べ方

C 言語には言語規格書なるものがあって、標準ライブラリ関数のプロトタイプや、関数がどのような動作をするのかも決められています。今やインターネットで検索すれば、これに準じるような文書が簡単に見つかります。

インターネットにつながってなくても、Unix 系（Cygwin を含む）なら man というコマンドがあって、`man 3 sin` を実行すると、sin 関数の説明が表示されます。（閲覧中の操作は、スペースキーで次のページ、`q` で終了です。）

man コマンドには各種の説明文（マニュアル）が集約されていて、セクション番号によって分類されています。“3” というのは C 言語のライブラリ関数を指し、Unix コマンドなら “1” にします。名前に重複がなければ省略しても構いません。詳細は `man man` で調べましょう。

7.1.3 プロトタイプ宣言における省略と重複

コンパイラが関数の型検査を行うときに、引数の**型**は重要ですが、引数の**名前**は関係ありません。ですから、プロトタイプ宣言では、引数の名前は何であってもエラーになりませんし、省略することすら可能です。しかも、矛盾さえなければ、同じ関数のプロトタイプ宣言を何度行ってもエラーになりません[*5]。以下は文法上、正しいプログラムです。

```
double square(double a); // プロトタイプ宣言（関数定義と引数名が異なる）
double square(double);   // プロトタイプ宣言（引数名を省略）
double square(double x); // プロトタイプ宣言（３度めでもOK）
double square(double x) { return x * x; } // 関数定義
```

しかし、このように引数名を省略したり、何度も宣言する積極的な理由は思いつきません。むしろ引数の名前からは役割が連想されて、関数を呼び出すプログラマの役に立ちそ

[*4] # で始まるので、プリプロセッサ命令です。コンパイルの最初期段階で処理されます。
[*5] 関数定義（本体）もプロトタイプ宣言の機能を含んでいますので、言語機能上、重複は禁止できません。

うです。したがって、これらのことは文法的な知識にとどめておいて、プロトタイプ宣言は単純に関数本体からコピーして、一度だけ記述すればよいでしょう。

コラム：テキストエディタの行選択とプロトタイプ宣言

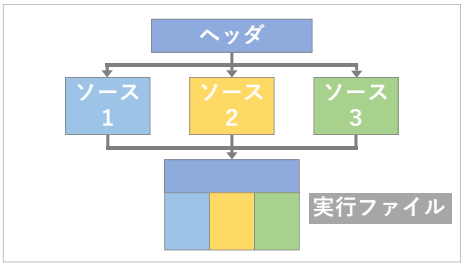
プロトタイプ宣言を書くには、テキストエディタで関数定義の**最初の 1 行**をコピーします。この作業に、マウス操作の得意 (?) な初学者が、**1 文字単位**で時間をかけて選択しているのをよく見かけますが、見ているだけでまだるっこしいです。

多くのエディタは、たとえマウスで操作しても、**行番号の数字の上をクリック**するだけで、(1 行の先頭から末尾まで）**行選択**されます。トリプルクリック（マウスのダブルクリックの、もう一度多くクリックする）でも簡単に選択できるでしょう。さらにキーボードなら 2〜3 操作（例えば Home Shift+↓）で 同様のことができますから、コピー＆ペーストは、間違えることなくすぐに終わります。普段からマウスでちまちま 1 文字単位で選択していると、関数の引数を修正するときに、面倒くさがってコピーし直さず、プロトタイプ宣言まで手作業で編集して、しかも間違えてコンパイルエラーで右往左往するという**絵に描いたような苦行**が待っています。書かずにすますこともできるプロトタイプ宣言で、わざわざ険しい道を行くわけです。

近頃は、選択した**領域の移動**（カット＆ペースト）ができずに、コピー＆ペースト＆**コピー元削除**という操作をする人が増えています。もちろん削除するのにコピー元をマウスで選択しなおすので、1 文字単位で範囲を間違えて、あっという間にコンパイルエラーです。早く「カット」と、操作ミスを元に戻す「アンドゥ」を覚えましょう。

テキストエディタにもよるのですが、行選択すれば、Tab と Shift+Tab でインデントの深さを選択領域まるごと調節できることがあります。ぜひ使いこなしたい機能です。

まずは 「カット」Ctrl+X、「コピー」Ctrl+C、「ペースト」Ctrl+V、そして「上書き保存」Ctrl+S、「アンドゥ」Ctrl+Z、「リドゥ」Ctrl+Y をマスターしましょう。これは最低限です。Home と End の行頭・行末ジャンプも重要です。そしてプロトタイプ宣言を書くのは、行選択が素早くできるようになってからです。



7.2 変数のスコープ・変数の寿命

MEMO: ヘッダでは宣言を行って、実体が伴わない。実体が伴うのは定義。

これまで、変数の定義は関数のブロックの中に書いてきました。これを**関数ブロックの外**（トップレベル）で行うとどうなるでしょうか。実はエラーにならずに、ちゃんと使えます。

この変数の**スコープ**（有効範囲）はどうなるでしょうか。さらに考えると、今まで気にならなかった、変数の**誕生や消滅**の瞬間がいつなのかが問題になってきます。順番に考えていきたいのですが、一つ頭の隅に覚えておいて欲しいことがあります。

本書のプログラムは、実行プログラムを 1 つのソースコードから生成しています。main 関数のある .c ファイルが 1 つです。しかし一般の C 言語プログラムは複数のソースコードで構成され、ファイルごとに別々にコンパイルする**分割コンパイル**が行われているということです。

7.2.1 ローカル変数・グローバル変数

関数の中で定義された変数のスコープは、**??**節で述べたように、関数のブロックの中だけでした。この変数は、スコープの狭さから、その関数の**局所変数**あるいは**ローカル変数** (local variable) と呼ばれます。もちろん、仮引数もローカル変数です。そして関数ブロックの入れ子になった内側のブロックで定義されていれば、スコープは**そのブロックのみ**と、より狭くなるのでした。

関数の外で定義された変数のスコープは、**プログラム全体**に広がります。手続き[*6]さえ踏めば、分割コンパイルされる別のファイルからでも参照できます。この変数は、スコープの広さから、**大域変数**あるいは**グローバル変数** (global variable) と呼ばれ、どの関数に属するかという概念がなくなります。

[*6] 本書では扱いません。関数にプロトタイプ宣言があるように、変数にも宣言のみ（実体を伴わない）があります。

表 7.1　スコープの種類

変数の種類	変数を定義する場所	実現されるスコープ	スコープの名称
グローバル	関数外（トップレベル）	プログラム全体	グローバルスコープ
static グローバル		1 つのソースファイル	ファイルスコープ
ローカル, static ローカル	関数内, 仮引数	1 つの関数	関数スコープ
	関数内のブロック内	関数内のブロック内	ブロックスコープ
	関数内のブロックのさらに内側の...	関数内のブロックのさらに内側の...	

7.2.2　時間的な有効期間＝寿命と static

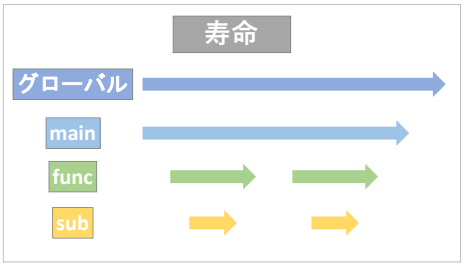
ここでは、プログラムの実行する部分が進むのに応じて、変数がどのタイミングで必要になり、どのタイミングで不要になるのかを見ていきます。つまり、変数の誕生から消滅までの、**寿命** (lifetime) を考えてみます。

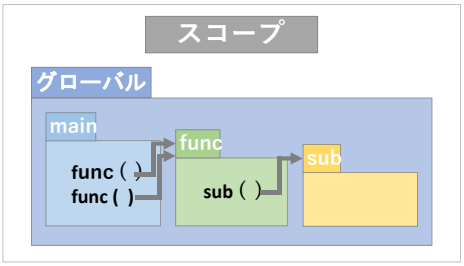
ローカル　ローカル変数は、定義されている**ブロックを実行**するときに誕生します。（記憶するための領域が確保されます。）そして、その**ブロックを抜け出す**ときに、不要になるので消滅します。（領域が開放されます。）ブロックをもう一度実行しなおせば、ローカル変数はまた新たに作り直されます。今まで気にすることはなかったかもしれませんが、このような動きになっていました。そして初期化は自動では行われず、プログラムで代入するまで、どんな値になっているのか保証がありません。

グローバル　グローバル変数は、いつどの関数から使われるかわかりませんから、消滅するわけにはいきません。**プログラム開始**と同時に誕生して、**プログラム終了**まで存続します。ローカル変数とは異なり初期値は 0 です[*7]が、0 であっても明示的に初期化（定義と同時に代入）するのがよいでしょう。

static **ローカル**　ローカル変数でも、計算結果を残しておくなど、次回の関数呼び出しまで値を保ちたいこともあるでしょう。そんなときは、ローカル変数の定義に `static`（静的）というキーワードをつけて「`static int a = 5;`」のように書きます。スコープが狭い特徴はそのままで、**グローバル変数相当の寿命**を獲得して、プログラム終了まで存続します。初期値もグローバル変数と同じく 0 です[*8]し、初期化（定義と同時に行う代入）もプログラム開始直後の 1 回だけになります。

[*7] ローカル変数と字面が同じで区別がつきにくいからでしょうか、あまり理解されていないようです。
[*8] static だと字面が違うためか、よく理解されているようです。





7.2.3 空間的な有効範囲＝スコープと static

7.2.1 項に続いて、もう一度**スコープ**、つまりプログラムの実行している場所とは無関係に、変数の定義のされかたによって、**参照を許す範囲**がどう変わるかをまとめてみます。

ローカル ローカル変数は、既に述べた通り、**ブロックの中**だけで有効です。（引数や関数本体のブロックで定義されるのものを関数スコープ、さらに内側のブロックならブロックスコープと区別したり、まとめてローカルスコープと呼ぶこともあります。）

グローバル グローバル変数は、やはり既に述べた通り、スコープが**プログラム全体**にわたります。（グローバルスコープと呼びます。）有効範囲があまりにも広いので、グローバル変数の**使用は最小限**にとどめるべしとの戒めとともに、たとえ使うとしても、変数名の重複が起こらないよう、**変数名を長く**したり、接頭辞や接尾辞で目立たせるなどの工夫が推奨されます。

static **グローバル** グローバル変数のスコープを少し狭くする方法があります。ここでも `static` をつけます[*9]。寿命は長くなりようがありませんが、スコープが**そのファイルだけ**に限定されます。（ファイルスコープと呼びます。）変数名が重複しないように気を配る範囲が、1 つのファイルだけでよくなります。

以上をまとめると表 7.1 のようになります。なお、スコープは変数以外にも考えられます。

関数 関数にも、どこから呼び出せるかというスコープを考えることができて、通常の関数はすべてのファイル、static をつけるとそのファイルだけになります。つまり**グローバル変数と同じ**です[*10]。

マクロ マクロのスコープは、**ブロックとは無関係**に、定義したところから、そのファイルの終わりまでです。static もつけられません。

[*9] C 言語は少ないキーワード（予約語）でやりくりしているので、文脈によって意味の変わることがよくあります。static も void に並んで意味のバリエーションの多いキーワードです。

[*10] 普通の関数がグローバルだとすると、ローカル関数（関数内の関数）が作れそうに思えるかもしれません。C 言語の規格にはないのですが、gcc のように、コンパイラの独自拡張で使える場合があります。

7.2.4　static の使用例

ソースコード 7.3 では、関数が呼び出されるたびに、1, 2, 3, . . . と、**呼び出された回数**を返す関数を作ろうとしています。4 通りの変数を使ってみました。

(A) `call_num_local()` では通常のローカル変数を使いました。関数呼び出しが起こるたびに n が 0 で初期化されるため、**返す値は常に 1** となり、目的の動作をしてくれません。関数の処理が終わった後でも、値を保持する変数が必要になります。

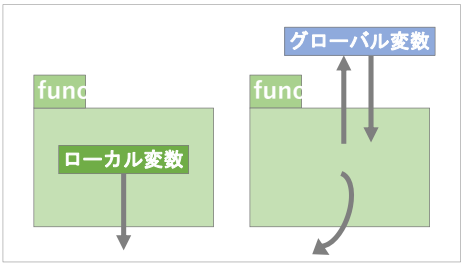
(B) `call_num_global()` ではグローバル変数を使いました。グローバル変数はスコープが広いので、この変数を使う関数名を接頭辞にして、長い変数名にしてみました。初期化（0 の代入）が行われるのはプログラム開始直後の 1 回だけで、関数の処理が終わっても**値を保持し続けて**、望みの動作をしてくれます。ひとまず目標達成です。しかし、変数名をいくら長くしたところで、プログラムの別の場所から値を（意図せず）変更してしまう危険が付きまといます。

(C) `call_num_static_local()` ではローカル変数に static 属性をつけてみました。変数のスコープはローカル変数と同じで、この関数専用です。変数名が短くても安全です。変数の寿命は static によって**グローバル変数と同等**になり、0 の代入もプログラム開始直後の 1 回だけ起こります。望みの動作をしてくれますし、変数名も短くできるので、この方法が一番適しているでしょう。

(D) `call_num_static_global()` ではグローバル変数に static 属性をつけてみました。変数のスコープはこのソースファイルのみと、少し狭くなるので、少し短い変数名にしました。今回は当てはまりませんが、**複数の関数**からアクセスする必要のある変数なら、この方法を採用します。

ソースコード 7.3 の実行結果

```
local=1, global=1, static_local=1, static_global=1
local=1, global=2, static_local=2, static_global=2
local=1, global=3, static_local=3, static_global=3
local=1, global=4, static_local=4, static_global=4
local=1, global=5, static_local=5, static_global=5
```



ソースコード 7.3 呼び出された回数を数える関数

```c
#include <stdio.h>

int call_num_local(void) { // (A) 失敗
    int n = 0; // ローカル変数（短い変数名）
    n++;
    return n; // 1, 1, 1, ...
}

int call_num_global_counter = 0; // グローバル変数（長い変数名）
int call_num_global(void) { // (B)
    call_num_global_counter++;
    return call_num_global_counter; // 1, 2, 3, ...
}

int call_num_static_local(void) { // (C)
    static int n = 0; // staticなローカル変数（短い変数名）
    n++;
    return n; // 1, 2, 3, ...
}

static int count = 0; // staticなグローバル変数（少し長い変数名）
int call_num_static_global(void) { // (D)
    count++;
    return count; // 1, 2, 3, ...
}

int main(void) {
    for (int i=0; i<5; i++) {
        printf("local=%d, ",         call_num_local());
        printf("global=%d, ",        call_num_global());
        printf("static_local=%d, ",  call_num_static_local());
        printf("static_global=%d\n", call_num_static_global());
    }
}
```

7.2.5　変数名や関数名の重複

これまで、2 つの変数の名前を同じにして、コンパイルエラーになったことがあると思います。**変数名や引数名は重複できない**ことを経験的に理解されたことでしょう。

この制限には、正確には「同一ブロックで」という但し書きがつきます。つまり内側にブロックを作れば、外側のブロックと同じ変数名でもよくなります。参照するときには、**まだ実行中のブロックのうち、一番内側**のものが優先して使用され、それより外側のブロックにある同名の変数は隠蔽され、参照できなくなります。（参照できなくなることを**シャドーイング** (shadow) といいます。）もっとも、同名の変数があるとややこしいので、避けたほうがよいでしょう[*11]。

```
    void x(void) { printf("x"); } // ここに関数がある(case A,B,C)
    void func(int x) {    // ここに引数があるので
//      int x=2;          // コンパイルエラー
        {
            int x=3;      // ブロックの内側なら OK
            {
                int x=4; // さらに内側のブロックなので OK
                printf("%d", x); // "4"
            }
            {   // このブロックにはxがないので直近のxを使う
                printf("%d", x); // "3"
            }
            printf("%d", x); // "3"
        }
        printf("%d", x); // "引数の値"
//      x();                        // コンパイルエラー(case C)
    }
//  int x=0;                        // コンパイルエラー(case B)
//  void x(void) { printf("x"); } // コンパイルエラー(case A)
```

ここで注意したいのは、C 言語では識別子（名前）に関して、**関数と変数は区別せず**に扱われる[*12]ことです。そして**関数**と**グローバル変数**は、スコープが同じであったことも思い出してください（☞7.2.3 項）。したがって、次のようになります。

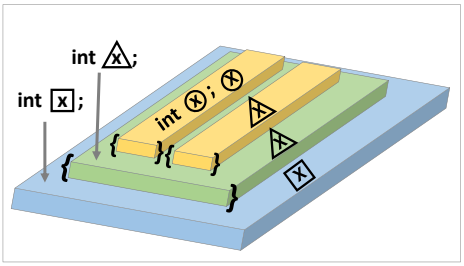
(case A) 同じ名前の関数を 2 つ作ることは（処理が同一でも）できません。

(case B) 関数と同じ名前のグローバル変数も作れません。

(case C) 関数と同じ名前のローカル変数を作ると、関数を呼び出せなくなります。

[*11] Java のようなオブジェクト指向言語では、クラスに直属のメンバ変数がシャドーイングされても参照する方法があって、わざと同名の変数（仮引数）にすることがよくあります。

[*12] C 言語では、関数は、特別な型の**変数**として扱われています。関数の引数に、関数を渡すことも可能です。



コラム： return に用いる変数名

戻り値を格納する変数名を、関数名と同じにする初学者を散見します。その関数自身を呼び出すこと（再帰呼び出し）がシャドーイングでできなくなるだけで、ほとんど実害はないのですが、関数名と変数名が重複しないよう心がけている身からすると、違和感を覚えてしまいます。

return に用いる変数名は、どんな関数でも**とりあえず ret** にしている人もいます。

C 言語と同時期に誕生した Pascal という言語には return 文がありません。戻り値は、**関数と同名の変数**に代入することで示します。初学者が自然にやりそうなことを言語規格に取り入れてあったのだと気づくと、この仕組みを思いついた人の洞察力には感歎してしまいます。

コラム： 複数の値を返す関数?

C 言語の関数は、引数ではたくさんの値を受け取ることができるのですが、返す値は最大でも 1 つだけに制限されています。コンパイラの設計を楽にするための、一種の割り切りとも思えます。

しかし現実には、複数の値を返したい場面もあります。そのような時のプログラム上の工夫がいくつかありますが、多くは今後学ぶ知識が必要になるので、今の段階では、そのような方法もあるのだ、ぐらいにとどめておいてもらえば構いません。

- 2 つの関数に分離する。
- 構造体を返す。（**??**章）
- 変数の配列（ポインタ）を受け取って、関数側で値を書き換える。（**??**章）

次の方法では失敗しますので、注意が必要です。（ヒント：寿命）

- 関数で確保したローカル変数の配列（ポインタ）を返す。（**??**章）

表 7.2　単語の連結戦略

名称	スネークケース (snake case)	キャメルケース (camel case)
方法	単語の間に_をはさむ	途中の単語の先頭が大文字
例	snake_case get_string_length	camelCase getStringLength
使用例	（小文字）C 言語 （大文字）各言語の定数	Java, C# 先頭大文字の使い分けあり

7.3　変数や関数の命名規則や習慣

変数名や関数名（識別子）には**「名は体を表す」**よう、名前と役目を一致させるのが大前提です。例えば、変数の tmp が幅広くやり取りされるのは、避けるべきです。

命名上の制約もあります。多くのコンピュータ言語では、識別子に**スペースもハイフンも**使えません。そこで、複数の単語の連結方法に工夫が凝らされています。この習慣は言語によっても異なります[*13]。世間でよく使われるのは、次の 2 通りです。

スネークケース (snake case)　`get_string_length` のように、**単語の間にアンダースコア**（_）をはさみます。C 言語ではこれが主流で、変数と関数はすべて小文字、マクロはすべて大文字にする習慣があります。ほかの言語でも、**定数には大文字**のスネークケースがよく用いられます。

キャメルケース (camel case)　`getStringLength` のように、途中の**単語の先頭 1 文字だけ大文字**にします。Java や C# でよく使われます。先頭の 1 文字も大文字にした**アッパーキャメルケース**に異なる意味を持たせる場合もあります。

もちろん、この規則が素直に適用できない場合もあります。GCD のような元から大文字の単語を使おうとすれば、get_GCD か get_gcd かどちらにしてよいのかもわかりませんから、その場その場で柔軟に考えればよいでしょう。

並べる単語の順番は、「動詞 + 目的語」[*14]にすることが多くなってきました。論理型なら「is+ 形容詞」「has+ 過去分詞」などにする習慣（☞**??**節）があります。

長い単語の省略パターンとして、initialize を init とするのは業界標準的です。number を num とするのは、本書でも採用していますが、時代とともに省略しない場面も増えて

[*13] その言語の標準関数の命名規則に影響されているのでしょう。
[*14] オブジェクト指向言語の習慣でもあります。

表 7.3 変数名の例

悪い名前の例	問題点	良い名前の例	改善点
× flag	いつ TRUE になるのか?	○ is_first	最初なら TRUE
× check_ok	何を調べている?	○ is_valid	有効かを調べる
× table	何を変換する?	○ binary2ascii	内部表記を表示用に変換
× no_data	否定は !no_data	○ exist_data	否定は !exist_data

きています。first ↔ last、max ↔ min、upper ↔ lower のような、よく使われる**対義語のペア**もあります。

ちょっと変わったところでは、deg2rad のように、変換するものの名前に現れる “2” は、数字 (two) の意味ではではなく、発音の同じ**“to” の意味を借用**しています。

逆に避けるべきといわれているのは、flag や table のように、いつ TRUE になるフラグ（論理型変数）か、何を変換するテーブル（対応表）なのかわからない名前です。良い名前は、例えば is_first とすれば最初が TRUE と誰にでもわかります。binary2ascii とすれば内部表記（バイナリ値）を表示用に変換するテーブルだろうと想像できます。

no_data のような**否定を含む論理型**の名前もやめておきましょう。逆の条件が !no_data と、二重否定になって読みにくくなります。真偽を逆にして exist_data の名前にしておくと、否定が一度ですみます。

このような習慣は、たくさんあります。他人のプログラムを読みながら、少しずつ覚えていきましょう。

7.4　関数の使い道

??節で、関数を作る目的を説明しました。関数には、手続きに名前をつける、「入力」と「出力」を明確にする、といった役割がありました。

関数は他にも使い道があります。**同じ結果になる異なる手続き**を、2 つの関数として作っておいて、動作を比較する（同じことを確かめる）と**信頼性**が上がります。プログラムを**高速化**する際にも、同じ動作をする関数を作って差し替えることがあります。

対になる処理を規則的に作ることもあります。例えばソースコード 7.4 は長さの単位のインチとセンチメートルで、互いに変換する 1 組の関数です。両方の関数で変換すると、元の値に戻るので、動作検証に好都合です。**必要なのが片方だとしても**、両方作ります。

ソースコード 7.4　インチとセンチメートルの相互変換

```
1  #include <stdio.h>
2  #define CM_PER_INCH 2.54  // 1inch = 2.54cm
3
4  double inch2cm(double cm)   { return cm   * CM_PER_INCH; }
5  double cm2inch(double inch) { return inch / CM_PER_INCH; }
6
7  int main(void) {
8      double inch  = 10.0;
9      double cm    = inch2cm(inch);
10     double inch2 = cm2inch(cm);
11     printf("%g(inch) = %g(cm) = %g(inch)\n", inch, cm, inch2);
12 }   //       10(inch) = 25.4(cm) = 10(inch)
```

本書でも折りに触れて例を示しますので、徐々に身につけていきましょう。

7.5 練習問題

1. **[仮引数のスコープ（☞??節）]**
   ??ページのソースコード**??**について、5 行目の triangle_side() 関数の仮引数を (double x, double y, double z) に変更したとき、動作が変わらないように**最小限**の修正を行え。ただし、4 行目のコメント中の「a,b,c」は「x,y,z」に修正する。これ以外に、実行結果に影響しない修正は行わない。
2. **[ループ vs. 公式、信頼性（☞??項、??節、??節）]**
   1 から n (> 0) までの整数の和 $\sum_{k=1}^{n} k$ を求めたい。次の 2 通りの方法で実装せよ。
   - `int sum(int n)` はループ処理で積算せよ。
   - `int sum2(int n)` は等差数列の和の公式 $\frac{n(n+1)}{2}$ を用いよ。

   この 2 つの関数の値が n の**奇数**のとき、**偶数**のとき、あるいは 50000 のような**大きな値**のときに一致するかを調査せよ。そして、同じ結果を返すはずの関数を複数の方法で実装することが、信頼性向上にどのように役立つかを考察せよ。
3. **[信頼性]**
   正の整数 x, y の最大公約数を 2 通りの方法で求めたい。（**??**ページのソースコード**??**を参照してもよい。）
   - `int gcd(int x, int y)` は、ユークリッドの互除法で実現せよ。
   - `int gcd2(int x, int y)` は、共通する約数の最大値をループ処理で求めよ。
4. **[1 組の値を返すペアの関数（☞13 ページのコラム「複数の値を返す関数?」）]**
   分数の足し算 $\frac{b}{a} + \frac{d}{c}$ の結果を表す分数 $\frac{bc+ad}{ac}$ を、**約分**した状態で得たい。簡単のため $a, b, c, d > 0$ とする。
   C 言語の関数の戻り値は 1 つだけなので、分数を表すためにペアで使う 2 個の関数を作ることにする。つまり、結果を表す分数の、
   - 分子を返す `int add_numer(int a, int b, int c, int d)`
   - 分母を返す `int add_denom(int a, int b, int c, int d)`

   を作れ。これらの関数は、それぞれ $bc + ad$ と ac を、この 2 つの最大公約数で割ってから返せばよい。最大公約数を得るのに 3. で作った関数を用いてよい。
5. **[プロトタイプ宣言、getter/setter（☞7.1 節、7.2.4 項）]**
   以下のプログラム（大きなプログラムの一部分）を**指示**に従って完成させ、**目標**を達成するための、プログラム中での **2 つの変数への禁止すべき操作**を述べよ。
   背景 作ろうとしている大きなプログラムは、指定された年月に関する処理をする。
   目標 プログラム全体で共通する年と月を保持したい。ただし、個々の関数で範囲外のエラー処理を省くため、常に定められた範囲内の値にしておきたい。

指示 年と月のそれぞれについて、以下の関数をそれぞれ作れ[*15]。

(get_???) 保持している値を返す。（**必ず定められた範囲内の値**を返すこと。）

(set_???) 引数で指定された値を保持する。（範囲外なら何もしない。）

そして、この 4 つの関数の**プロトタイプ宣言**を書き加えよ。

範囲 年は 1970 から 2100 の整数をとる。月は 1 から 12 の整数をとる。

```
static int year  = 2000;
static int month = 1;

/* getter */
int get_year(void)  { return year; }
int get_month(void) { /* ここを作る */ }

/* setter */
void set_year(int y)  {
    if (1970 <= y && y <= 2100) { year = y; }
}
void set_month(int m) { /* ここを作る */ }
```

6. **[関数の役割分担・逆関数の作成（☞??節、??節）]**

   次の関数を作れ。（標準ライブラリの三角関数については**??**節を参照し、角度をラジアンで扱うことに注意せよ。180（度）が π（ラジアン）に対応する。）

   - `double deg2rad(double deg)` は deg（度）をラジアンに変換する。
   - `double rad2deg(double rad)` は rad（ラジアン）を度に変換する。

   円周率 π を表す定数は、C 言語規格には存在しないが、コンパイラの独自拡張でサポートされ、典型的には <math.h> でマクロ定数 `M_PI` が定義される[*16]。

   - `double sin_deg(double deg)` は deg（度）の正弦を返す。
   - `double asin_deg(double x)` は x の逆正弦を度数法（度）で返す。

   そして、逆の動作をする関数を、片方ずつ別々に作るのに比べ、同時に 2 個作った場合のデバッグの効率性や、作られた関数の信頼性について考察せよ。

7. **[関数の役割分担]** ☞**??**項

[*15] 一般に、内部で使う変数の、値を取り出す関数を getter、値を保存する関数を setter と呼びます。慣用句としては、この 2 つを対にして作ります。

[*16] Visual C++ では `#include <math.h>` の前に「`#define _USE_MATH_DEFINES`」の必要な場合がある。逆に gcc の `-std=c99` オプションは独自拡張をやめるので、`M_PI` が使えなくなる。

索引