Ⅱ 基本プログラミング

1．スケルトンプログラム

(1) hello world

［学習項目］①エディタの基本操作、②コンパイル操作、③printf () 関数と¥a、¥b、¥f、¥n、¥r、¥t、¥v、¥¥、¥?、¥'、¥"、¥ooo、¥xhhのエスケープ文字列

①エディタの基本操作

ごく簡単なプログラムを書いてみる、簡単だといってもC言語の作成者であるカーニンハン先生とリッチー先生が書かれたC言語についての最初の本であるK&R本に最初の例題として提示されている由緒正しいプログラムである。

```
#include <stdio.h>                    1行目
void main(void)                       2行目
{                                     3行目
    printf("hello, world¥n");         4行目
}                                     5行目
```

各行末の1行目～5行目は解説用に付けたものであるから入力しない。入力できたら適当なファイル名を付けてセーブする。ここではhello.cとでもしておく。C言語の場合改行やタブレーションは、強い意味を持たず、フリーフォーマットで、このプログラムは次のように記述しても全く同じように動作する。時間が許せばエディタの操作練習を兼ねて生成された実行ファイルの動作が同じであることを確認しておく。

```
#include <stdio.h>
void main(void){printf("hello, world¥n");}
```

C言語はフリーフォーマットであり、各ステートメントの区切りはセミコロンだけなのでこのような記述が可能になる。しかし、コンパイラにとって問題なくても、プログラムを読みメンテナンスを行うのは人間であるから、上記のように適切に改行や中括弧で囲まれたブロックを明確にするために用いるタブレーションによるインデント等は付けるようにする。

②コンパイル操作

実習で使用するコンパイラに応じてオプションを指定してコンパイルする。ここではgccをccで起動するとして例示する。

```
>cc -o hello hello.c
```

実行結果

```
>hello
hello, world
>
```

③ｐｒｉｎｔｆ（）関数

１行目は、標準入出力関数を使うための準備である。この行のように#で始まる行は、Ｃ言語のプリプロセッサに対する指示を意味し、行の後ろの方に「；」セミコロンがつかず、プリプロセッサによって置き換え処理をされてからＣ言語のコンパイラに渡されるので、Ｃコンパイラの目に触れることはない。Ｃのステートメントではないのだが実際の作業を考えたとき#で始まるプリプロセッサの働きはかなり大きいといえる。２行目はここからこのプログラムのｍａｉｎ（）関数の定義が始まるという意味である。元来関数名は任意であるが、一番最初に呼び出される関数の名前だけはｍａｉｎに固定されている。ここでｍａｉｎ（）の前のｖｏｉｄはこの関数の戻り値がないことを表し、ｍａｉｎ（）の小括弧の中のｖｏｉｄは、この関数は引数を持たないことを表す。

戻り値がないのがｖｏｉｄなので、このプログラムには戻り値を返すためのｒｅｔｕｒｎ文が含まれていない。何とＣ言語では「ない」に対してもデータタイプの規定があるのである。全ての存在（名前）は明確なデータタイプを持たねばならないのがＣ言語の大原則である。

３行目と５行目の中括弧は対になっており、ｍａｉｎ（）関数を定義しているブロックの始まりと終わりを表している。

このプログラムの実行内容は、４行目のｐｒｉｎｔｆ（）関数の呼び出しだけである。ｐｒｉｎｔｆ（）関数は、代表的な出力関数でＣ言語の学習に際して必ず最初の方に登場する。関数としては１個以上の引数を持つやや特殊なタイプの関数である。ｐｒｉｎｔｆ（）関数の戻り値は標準出力への転送バイト数を表し、戻り値が負の値の場合出力に失敗したことを意味する。

１個以上の引数を持つが、最初の引数は文字列リテラル（文字列定数）が想定されている。タイプで言えばｃｈａｒ型のポインタである。コンパイルして生成された実行可能ファイルを実行してみるとよく分かるが、文字列リテラルの内容がそのまま表示されているように見える。しかしよく観察すると”（ダブルクオーテイション）と￥ｎが表示されていない。”は文字列の始まりと終わりを示し、ここでは”と”ではさまれた部分が表示対象となる。￥ｎについては￥ｎを削除したプログラムを実行して確かめてみる。

```
#include <stdio.h>                    1行目
void main(void)                       2行目
{                                     3行目
    printf("hello, world");           4行目
}                                     5行目
```

４行目の￥ｎを削除する。削除操作以外は先程の復習である。

```
>cc -o hello1 hello.c
```

実行結果

```
>hello1
hello, world>
```

実行結果は大体同じであるが先の実行例では次の行にあった入力促進プロンプト（＞）がｗｏｒｌｄの後ろに付いてきている。￥ｎは、この二文字で改行を意味する。ＢＡＳＩＣのように改行抑制のコードを付加しないと勝手に改行してしまう言語もあるが、コンピュータ内部では改行も特殊な文字として扱われており、Ｃ言語のこの記述はコンピュータ内部の動作に正確に対応したものといえる。

￥ｎは改行を示すエスケープ文字列でコンピュータのメモリ上では１バイトのコードに対応することになる。日本語システムでは￥キーとの組み合わせで表現するが、￥キーは英語システムをはじめとする日本語以外のシステムでは存在せず同じアスキーコードに＼キー（バックスラッシュ）が対応させられているので＼ｎが改行を表す。

エスケープ文字列は改行のように、文字として表示されないが端末の動作に影響を与えるコードを表現するのに用いられる。エスケープ文字列には改行を示す￥ｎ以外に￥ａ（警告・ベル文字）、￥ｂ（バックスペース）、￥ｆ（フォームフィード・改頁）、￥ｒ（復帰）、￥ｔ（水平タブ）、￥ｖ（垂直タブ）、￥￥（円記号・バックスラッシュ）、￥？（疑問符）、￥’（単一引用符・シングルクオーテイション）、￥”（二重引用符）、￥ｏｏｏ、￥ｘｈｈがある。

￥ｎなら改行キー、￥ｂならバックスペースキー、￥ｔなら（水平）タブキーというふうに対応するキーを押したのと同じ影響を画面の出力に及ぼし、￥ｆ（フォームフィード・改頁）や￥ｖ（垂直タブ）￥ｒ（復帰）はプリンタの出力で見るとその動作が明確である。￥ａ（警告・ベル文字）は端末のビープ音等を出力する働きを示す。文字の表示位置等に影響は与えない。

それに対して￥￥（円記号・バックスラッシュ）、￥？（疑問符）、￥’（単一引用符・シングルクオーテイション）、￥”（二重引用符）は、Ｃ言語のプログラムの記述に際して何らかの意味付けされたシンボルで、通常そのままでは出力できないキャラクタの出力に用いる。

￥ｏｏｏと￥ｘｈｈは１バイトコードの８進表記や１６進表記を表す。文字列の末尾を示すのに用いられるヌル文字・空文字は￥０で表現される。￥ｘ３０と￥０６０はともに数字の０を表し、￥ｘ４１と￥１０１はともに大文字のＡを表し、￥ｘ６１と￥１４１はともに小文字のａを表す。￥ｂは￥ｘ０７や￥００７と同じで￥ｔは￥ｘ０９や￥０１１と同じである。

以下のようなプログラムの動作を推測しその後その動作を確認しておく。＃ｉｎｃｌｕｄｅ＜ｓｔｄｉｏ．ｈ＞の記述は含まれているものとする。

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｐｒｉｎｔｆ（”ＬＩＮＥ１￥ｎ”）；
  ｐｒｉｎｔｆ（”ＬＩＮＥ２￥ｎ”）；
}
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
```

```
  printf ("LINE1¥nLINE2¥n");
}

void main(void)
{
  printf ("LINE1");
  printf ("LINE2¥n");
}

void main(void)
{
  printf ("LINE1¥t");
  printf ("LINE2¥n");
}

void main(void)
{
  printf ("LINE1¥tLINE2¥n");
}

void main(void)
{
  printf ("LINE1¥x09");
  printf ("LINE2¥n");
}

void main(void)
{
  printf ("Beep¥n");
  printf ("¥a");
}
void main(void)
{
  printf ("Beep¥7¥n");
}
```

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｐｒｉｎｔｆ（”￥ｘ３０￥ｘ３１￥ｘ３２￥ｎ”）；
  ｐｒｉｎｔｆ（”￥ｘ４１￥ｘ４２￥ｘ４３￥ｎ￥ｘ６１￥ｘ６２￥ｘ６３￥ｎ”）；
}
```

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｐｒｉｎｔｆ（”￥￥　￥？　￥’￥”￥ｎ”）；
}
```

プログラムリストで￥と￥の間、？と￥の間、’と￥の間にスペースを入力しておき下記のプログラムの実行結果と対比させスペースは画面に何も表示されないが一文字であることを確認しておく。

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｐｒｉｎｔｆ（”￥￥￥ｘ２０￥？￥ｘ２０￥’￥ｘ２０￥”￥ｎ”）；
}
```

［拡張表記の例］

16進表現	表記	コメント	意味
0x07	¥a	alert	警告
0x08	¥b	backspace	後退
0x09	¥t	horizontal tab	水平タブ
0x0a	¥n	new line	改行
0x0b	¥v	vertical tab	垂直タブ
0x0c	¥f	form feed	書式送り
0x0d	¥r	carriage return	復帰
0x00	¥0	null	空文字
0x22	¥"	double quotation marks	二重引用符
0x27	¥'	single quotation marks	一重引用符
0x3f	¥?	question mark	疑問符
0x5c	¥¥	backslash	円記号

漢字の扱いについて少し触れておく。漢字は実は内部的に2バイトで一文字を表現しており、コード体系もパーソナルコンピュータではシフトＪＩＳが主流であるが、Ｗｉｎｄｏｗｓ－ＮＴでは内部はＵＮＩコードで処理しシフトＪＩＳで見せるような動作をしている。ＵＮＩＸでは、シフトＪＩＳに加えてＪＩＳやＥＵＣも切り替えて扱うことが可能である。アルファベットや数字のアスキーコード体系のように、基本的にほとんど共通のコード体系が一つ存在するという状態にはなっていない。でありからＣ言語のプログラム中における漢字の扱いについては若干様子をみる必要がある。幸い二種の試験では漢字の扱いについて問われる事はない。また文字列リテラル中への漢字の埋め込みは大多数のコンパイラでは何ら問題を起こさず、大多数のシステムでは文字定数として漢字を一文字として処理することが可能である。

また／＊と＊／の記述は、コメント扱いになりＣコンパイラに無視される。コメントはプログラマのための記述（メモ）であるが、コメント中の漢字の利用についても大多数のコンパイラにおいて制限は存在しない。以下に例を挙げる。

```
#include <stdio.h>
/* 初めてのCプログラミング */
void main(void)
{
    printf("hello, world¥n");
}
```

（2）printf変換とscanf変換

［学習項目］①変数の宣言、②書式付き出力ｐｒｉｎｔｆ()、③書式付き入力ｓｃａｎｆ()

```
#include <stdio.h>                                    1行目
void main(void)                                       2行目
{                                                     3行目
  int x;                                              4行目
  printf("x="); scanf("%d", &x);                      5行目
  printf("x=%d(10)¥t%x(16)¥n", x, x);                 6行目
}                                                     7行目
```

標準入力より入力された０～９の範囲の数字列（符号付きで宣言しているので＋・－の符号も入力可能である。）を１０進数として受け取り、受け取った内容を１０進数と１６進数で表示するプログラムである。４行目で整数型の符号付き変数ｘを宣言し５行目のｓｃａｎｆ()関数でキーより入力された内容を整数の１０進数として変数ｘに設定する。６行目のｐｒｉｎｔｆ()中の％ｄで１０進数として、％ｘで１６進数として表示する。

①変数の宣言

ｉｎｔ、ｃｈａｒは整数値を格納するための入れ物・変数の宣言に用いる。ｆｌｏａｔ、ｄｏｕｂｌｅは実数を格納するための変数の宣言に用いる。１文字は８ビット長の整数値として格納される。ｃｈａｒ、ｉｎｔは内部構造を持たないが、ｆｌｏａｔ、ｄｏｕｂｌｅは指数部と仮数部に対応した内部構造を持つ。これら４つの基本的なデータタイプに付加する形でｓｉｇｎｅｄ、ｕｎｓｉｇｎｅｄ、ｓｈｏｒｔ、ｌｏｎｇの修飾子を用いる事もある。ｓｉｇｎｅｄ、ｕｎｓｉｇｎｅｄは符号付き、符号なしの指定に用いられ、ｓｉｇｎｅｄ指定は最上位ビットを符号ビットと見なすことを表し、ｕｎｓｉｇｎｅｄの指定は最上位ビットも値を表現するのに用いる事を示す。ｓｉｇｎｅｄ、ｕｎｓｉｇｎｅｄの修飾子を付加しない時は、ｓｉｇｎｅｄが省略されたものと見なされる。ｓｈｏｒｔ、ｌｏｎｇは変数・入れ物のサイズの拡張に用いる。処理系によって差があるが、ｓｈｏｒｔ修飾子を付けた場合と付けない場合あるいはｌｏｎｇ修飾子を付けた場合と付けない場合で差が生じず、同じサイズの変数が生成されるコンパイラも多い。

ｖｏｉｄは、サイズが０であることを示す。副作用だけを利用する関数の戻り値の宣言にはよく用いるが、通常ポインタ型でない変数の宣言には用いない。

ｉｎｔ　ｘ，ｙ，ｚ；のようにカンマを用いることによって同じタイプの変数を一度に宣言することが可能である。これはｉｎｔ　ｘ；ｉｎｔ　ｙ；ｉｎｔ　ｚ；のように３つに分けて宣言されたのと全く同じ結果を生む。

変数に値を設定する場合、代入操作を示す＝（イコール）を用いてｘ＝３；のように記述する。変数の用い方によっては、宣言時０クリアしておきたい場合やデフォルトの値を設定しておきたい場合がある。そのような場合、宣言と代入を分けて記述しても良いが、ｉｎｔ　ｓ＝０；のように宣言と同時に値を設定することができる。Ｃ言語ではこのように初期化付きの宣言が可能である。もっとも独立した代入文のイコールの右辺には任意の式が記述可能であるが、宣言時に初期化する場合は定数式に限られる。また変数ｋを定数を持つ変数として用いる場合、ｃｏｎｓｔで修飾しｃｏｎｓｔ　ｉｎｔ　ｋ＝３；のように記述すると変数ｋは以後プログラム中での変更を受け付けなくなる。以下に初期化つき宣言の例を示す。

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｉｎｔ　ｘ＝１０，ｙ＝２１，ｚ＝３２；
  ｐｒｉｎｔｆ（”ｘ＝%３ｄ　ｙ＝%３ｄ　ｚ＝%３ｄ¥ｎ”，ｘ，ｙ，ｚ）；
}
```

実行結果

```
ｘ＝　１０　ｙ＝　２１　ｚ＝　３２
```

変数は使う前にデータタイプを指定した宣言が必要で、宣言を記述する場所は基本的に{～}ブロックの先頭部分である。ｉｎｔ　ｘ＝１０，ｙ＝２１，ｚ＝３２；は良いがｉｎｔ

ｘ；ｘ＝１０；ｉｎｔ　ｙ；ｙ＝２１；ｉｎｔ　ｚ；ｉｎｔ　ｚ＝３２；の順番に記述すると変数ｙとｚの生成がうまくいかない。これは｛～｝ブロックの先頭に連続して変数の宣言を行う必要があるためである。ここではｘ＝１０；の代入操作が後に続くｉｎｔ　ｙ；ｉｎｔ　ｚ；の宣言を妨害している事になる。上記の例で変数の宣言と値の設定を分離して記述する場合ｉｎｔ　ｘ；ｉｎｔ　ｙ；ｉｎｔ　ｚ；ｘ＝１０；ｙ＝２１；ｚ＝３２；のように宣言は連続して記述する必要がある。

②書式付き出力ｐｒｉｎｔｆ（）

先に学習したようにｐｒｉｎｔｆ（）関数の第一引数である文字列リテラル中の￥は後に続く文字と２個一組になって制御コードを示す働きをする。％は￥と同じように後に続く文字列が特殊な働きをする事を示す。このプログラムの５行目では、％ｄが第２引数の式の評価値を１０進数で表示することを示し、％ｘが第３引数の式の評価値を１６進数で表示することを示す。５行目では第２引数の式も第３引数の式もともにｘという単項式であるが、これはたまたま同じ式が置かれているだけであり、同じ内容だからといって１個だけですませることはできない。文字列リテラルの左から数えて最初の％には第２引数の式の評価値が、２番目の％には第３引数の評価値が、ｎ番目の％には第（ｎ＋１）引数の評価値が対応する。なお％を表示したい場合は％％と書けば％一文字に置き換わって表示される。

％ｄと同じように％ｉも整数値を１０進数で表示する。％ｘは％ｘと％Ｘのようにｘが大文字でも小文字でも整数値を１６進数で表示することには変わりがないが、％ｘのように小文字の場合１０～１５に対応するａ～ｆが小文字で表示され、％Ｘのように大文字の場合１０～１５に対応するＡ～Ｆが大文字で表示される。％ｏは８進数で表示することを示す。％ｘ、％Ｘ、％ｏは符号なしの表示となる。％ｄや％ｉは基本的に符号付きで、符号なしで１０進表示をする場合％ｕを用いる。

ｃｈａｒ型の値は整数値としても表示可能であるがその場合アスキーコード表で各文字に対応する整数値が表示される。ｃｈａｒ型の整数値に対応する文字を表示する場合％ｃを用いる。ｃは小文字である。

変換指定子

整数型に対応

ｄ変換、ｉ変換	ｄｅｃｉｍａｌ　ｉｎｔｅｇｅｒ
	符号付き１０進表記
ｏ変換	ｏｃｔａｌ
	無符号８進表記
ｕ変換	ｕｎｓｉｇｎｅｄ
	無符号１０進表記
	ｕｎｓｉｇｎｅｄ　ｉｎｔ　実引数
ｘ変換	ｈｅｘａｄｅｃｉｍａｌ

無符号１６進小文字表記　０～９ａ～ｆを用いた表記

Ｘ変換　　ｈｅｘａｄｅｃｉｍａｌ

無符号１６進大文字表記　０～９Ａ～Ｆを用いた表記

浮動小数点型に対応

ｆ変換　　ｆｌｏａｔｉｎｇ　ｐｏｉｎｔ

１０進浮動小数点表記

ｅ変換、Ｅ変換　ｅｘｐｏｎｅｎｔ

１０進指数表記　　～ｅ+/-～、～Ｅ+/-～

文字型に対応

ｃ変換　　ｃｈａｒａｃｔｅｒ

文字列に対応

ｓ変換　　ｓｔｒｉｎｇ

ポインタ型に対応

ｐ変換　　ｐｏｉｎｔｅｒ

ｄｏｕｂｌｅ型やｆｌｏａｔ型の実数型は内部構造を持つため、％ｄや％ｘではうまく表示できない。％に続く指定はＰｒｉｎｔｆ変換であって、型のキャスティング・変換は行わない。実数型を表示する場合は、％ｆ、％ｅ、％Ｅ、％ｇ、％Ｇを用いる。以下に実数型の表示の例を示す。

scanf("%f",float)
scanf("%lf",double)
printf ("%f", { float / double })
} である。

```
#ｉｎｃｌｕｄｅ　＜ｓｔｄｉｏ．ｈ＞　　１行目
#ｄｅｆｉｎｅ　ＰＩ　３．１４　　２行目
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）　　３行目
{　　４行目
　ｄｏｕｂｌｅ　ｒ，ｓ；　　５行目
　ｒ＝１０．５；　　６行目
　ｓ＝ｒ＊ｒ＊ＰＩ；　　７行目
　ｐｒｉｎｔｆ（”半径%ｆの円の面積は%６．１ｆです。￥ｎ”，ｒ，ｓ）；
}
```

実行結果

半径　１０．５　の　円の面積は　３４６．１　です。

２行目の#ｄｅｆｉｎｅ　ＰＩ　３．１４　は、#ｉｎｃｌｕｄｅと同じように#で始まるプ

リプロセッサに対する指示である。プリプロセッサは、７行目のＰＩを３．１４に置き換えてｓ＝ｒ＊ｒ＊３．１４；にしてからＣコンパイラにプログラムを渡すので、Ｃコンパイラにとっては２行目はあってもなくても同じである。しかしこのプログラムを読む人にとっては３．１４が円周率を示していることがより明確になる効果が期待できる。このようなプリプロセッサの置き換え機能を利用した定義がマクロ定義である。

５行目で倍精度浮動小数点型の変数ｒとｓを宣言している。宣言しただけでは指定したサイズの場所が確保され、変数名でアクセス可能になるだけで、変数の中に入っている値は不定である。６行目で変数ｒに半径の値を設定している、この行は数学の方程式と同じ様な記述になっているがその意味は非常に異なる。数学の方程式はこの未知数はこんな条件を満足する状態にあるという意味で、状態の記述である。それに対してＣ言語では、左辺の変数に右辺の式の評価値を設定する代入操作という、コンピュータに対する動作の指示を表す。この式は変数ｒの値を１０．５にする。変数ｒで指定されるメモリ中に１０．５という値をセットするという意味になる。

７行目の=の右辺は変数ｒを含む､変数名が=の左辺に置かれるときは、右辺の計算結果を格納する場所を示す。それに対して変数名が右辺に含まれる時その変数は評価される。評価されるといっても甲、乙、丙、丁のように成績をつけるわけではなく、変数の持っている値を取り出し、式中の変数名の場所にその変数の持つ値をあてはめて計算が進む事である。ここではｒの値が１０．５であるから１０．５＊１０．５＊３．１４の計算結果である３４６．１８５が評価結果となり、変数ｓには評価結果の３４６．１８５が格納される。

計算結果が変数中に保持されているだけでは我々には判読不能である。そこでｐｒｉｎｔｆ（）関数の出番となる。先に述べたようにｐｒｉｎｔｆ（）関数の２番目以降の引数は最初の引数の文字列中に含まれる%～に対応する形で置かれていく。ここではｒが最初の%ｆに対応し、ｓが２番目の%６．１ｆに対応する形で置かれている。%記号は、出力関数中では出力変換仕様を示しｓｃａｎｆ()のような入力関数では入力変換仕様の指定に用いる。ｆはｆｌｏａｔｉｎｇ　ｐｏｉｎｔのｆでｆｌｏａｔ型・実数型に対応する。%ｆで実数型の変数の内部フォーマットで値の組を読み出し、１０進表現で表示することを指定している。フォーマットを持つ場合フォーマットをきちんと指定してやらないと内部的に正しい値を持っていても正しく表示されない。

%とｆの間の６．１は表示精度の指定で、小数点を含む全体を６桁で表示し、その内小数点以下は１桁で表示せよという指定である。変数ｓは３４６．１８５という値を持ってるが、ここで小数点以下を１桁に制限しているので小数点２桁目以下は表示されず３４６．１と表示されている。

・ｃｈａｒ型の表示

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
```

```
  ｃｈａｒ　ｘ；
  ｘ＝’Ａ’
  ｐｒｉｎｔｆ（”変数ｘの内容は　％ｃ　です。￥ｎ”，ｘ）；
}
```

実行結果

変数ｘの内容は　Ａ　です。

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
 {
  ｃｈａｒ　ｘ＝０ｘ４１；／＊　０ｘ４１＝’Ａ’　＊／
  ｐｒｉｎｔｆ（”変数ｘの内容は　％ｃ　です。￥ｎ”，ｘ）；
}
```

実行結果

変数ｘの内容は　Ａ　です。

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
 {
  ｃｈａｒ　ｘ＝’Ａ’；
  ｐｒｉｎｔｆ（”変数ｘの内容は％ｃで、16進表記では％ｘになる￥ｎ”，ｘ，ｘ）；
}
```

実行結果

変数ｘの内容はＡで、１６進表記では　41になる

ｆｌｏａｔ型やｄｏｕｂｌｅ型の変数は内部にパラメータをもちそのフォーマットは複雑である。それに対しｃｈａｒ型とｉｎｔ型は、文字と整数で非常に異なるイメージがあるがどちらも内部フォーマットを持たず、その差異はサイズの違いだけである。であるからこのように１６進数表記で内部の値を読み出したり設定したりすることができる。。

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
 {
  ｃｈａｒ　ｘ＝’Ａ’；
  ｐｒｉｎｔｆ（”変数ｘの内容は％ｃで、16進表記では％ｘになる￥ｎ”，ｘ，ｘ）；
  ｘ＝’Ｂ’；
  ｐｒｉｎｔｆ（”変数ｘの内容は％ｃで、16進表記では％ｘになる￥ｎ”，ｘ，ｘ）；
}
```

実行結果

変数ｘの内容はＡで、16進表記では　41になる

変数ｘの内容はＢで、16進表記では　42になる

アスキーコード体系では、アルファベットは連続した値を割り振られていることがわかる。変数ｘに最初に’Ａ　’を設定して表示し、次に’B’を設定して表示している。このように=を用いて変数に値を設定する操作が代入操作で同じ変数に対して複数回代入操作を行った場合後から設定された値が有効になる。代入操作以前に変数に入っていた値を消して後から代入された値が設定されるので、破壊的代入操作がなされたという表現をする事もある。次のような練習問題をさせておくのも悪くないと思われる。

・小文字のａを変数ｙに代入し、’ａ’に対応する値を16進数で表示するプログラムを作成せよ。

・数字の１と０に対応する値を16進数で表示するプログラムを作成せよ。

・スペースのアスキーコード表上の値を16進数で表示するプログラムを作成せよ。

［プログラミング例］

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｃｈａｒ　ｙ＝’ａ’；
  ｐｒｉｎｔｆ（”変数ｙの内容は%ｃで、16進表記では%ｘになる￥ｎ”，ｙ，ｙ）；
}
```

実行結果

変数ｙの内容は　ａで、これを16進表記すると　61になる。

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｃｈａｒ　ｘ＝’１’；
  ｐｒｉｎｔｆ（”変数ｘの内容は%ｃで、16進表記では%ｘになる￥ｎ”，ｘ，ｘ）；
  ｘ＝’０’；
  ｐｒｉｎｔｆ（”変数ｘの内容は%ｃで、16進表記では%ｘになる￥ｎ”，ｘ，ｘ）；
}
```

実行結果

変数ｘの内容は１で、16進表記では　31になる

変数ｘの内容は０で、16進表記では　30になる

注）数字も、アスキーコード表上では０から１～９の昇順に並んでいる。

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｃｈａｒ　ｙ＝’?’；／＊シングルコーテイションの間にスペースが入る　＊／
  ｐｒｉｎｔｆ（”変数ｙの内容￥’%ｃ￥’は%ｘ（１６）です。￥ｎ”，ｙ，ｙ）；
}
```

実行結果
変数ｙの内容’?’は　20（16）です。

・ｉｎｔ型の表示

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｉｎｔ　ｘ＝１７；
  ｐｒｉｎｔｆ（”変数ｘの内容は%５ｄです。￥ｎ”，ｘ）；
  ｐｒｉｎｔｆ（”変数ｘの内容は%－５ｄです。￥ｎ”，ｘ）；
}
```

実行結果
変数ｘの内容は　　１７です。
変数ｘの内容は１７　　です。

％５ｄの５は、最小フィールド幅の指定で最低でも５桁分の幅を用いての表示を表している。前出の％６．１ｆの場合６が最小フィールド幅の指定で．１が精度を指定している。Ｌは変換修飾子の一種でｌｏｎｇ　ｄｏｕｂｌｅ変換を表し、ｌｏｎｇ　ｄｏｕｂｌｅ実引数への対応を表す。ｈはｓｈｏｒｔ修飾でｓｈｏｒｔ　ｉｎｔ実引数への対応を表す。ｌ小文字のエルはｌｏｎｇ修飾でｌｏｎｇ　ｉｎｔ実引数に対応する。

％－５ｄの５は最小フィールド幅の指定であることに変わりはない。％と最小フィールド幅の指定の間に＋、－、空白文字（スペース）、＃や０（ゼロ）を書いた場合フラグを指定したことになる。－フラグは、左詰めフラグで表示桁数が最小フィールド幅に満たない場合、左詰で表示される。－フラグを指定しない場合は右詰で表示される。

＋は符号フラグで正の値を有符号化した場合、プラス符号を前に付けて表示する。符号フラグがなくてもマイナスの値はマイナス符号を付けて表示される。空白文字は空白フラグともいい符号フラグと逆の動作をする。正の値を有符号化した場合プラス符号の代わりにスペースを前に付けて表示する。

＃は表記フラグで８進表記を指定する変換指定子ｏが続いているなら先頭に０（ゼロ）を付加し、１６進表記を指定する変換指定子ｘが続いているなら０ｘ、変換指定子Ｘが続いているなら０Ｘを先頭に付加して表示する。０はゼロフラグで最小桁数に満たない部分を０詰めして表示する。

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｉｎｔ　ｘ＝１７；
  ｐｒｉｎｔｆ（”変数ｘの内容は%５ｘです。￥ｎ”，ｘ）；
```

```
  ｐｒｉｎｔｆ（”変数ｘの内容は%０５ｘです。￥ｎ”，ｘ）；
  ｐｒｉｎｔｆ（”変数ｘの内容は%＃５ｘです。￥ｎ”，ｘ）；
  ｐｒｉｎｔｆ（”変数ｘの内容は%＃５Ｘです。￥ｎ”，ｘ）；
  ｐｒｉｎｔｆ（”変数ｘの内容は%＃５ｏです。￥ｎ”，ｘ）；
}
```

実行結果

変数ｘの内容は　　１１です。

変数ｘの内容は０００１１です。

変数ｘの内容は　０ｘ１１です。

変数ｘの内容は　０Ｘ１１です。

変数ｘの内容は　　０２１です。

注）１７＝１６＋１＝１＊１６＋１＝２＊８＋１と表せる。

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｉｎｔ　ｘ＝－１；
  ｐｒｉｎｔｆ（”ｘ＝　%d［16進数：%４ｘ］￥ｎ”，ｘ，ｘ）；
}
```

実行結果

ｘ＝-１［16進数：ｆｆｆｆ］

注）ｉｎｔ型の値を１６ビット長とすると、ｆｆｆｆ＋１＝１００００（１６進数）で最上位の１がオーバーフローで消えるので１６ビット長の変数では１００００＝０００００＝０となり、ｆｆｆｆは１を足すと０になる値なので-１として扱うことができる。このようにＣ言語では有符号整数型では２の補数表現を用いている。

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｉｎｔ　ｘ＝０ｘ４３；
  ｐｒｉｎｔｆ（”変数ｘの内容は16進表現で　%ｘです。￥ｎ”，ｘ）；
  ｐｒｉｎｔｆ（”アスキーコード表で対応する文字は　%ｃ　です。￥ｎ”，ｘ）；
  ｘ＝’ｄ’；
  ｐｒｉｎｔｆ（”変数ｘの内容は16進表現で　%ｘです。￥ｎ”，ｘ）；
  ｐｒｉｎｔｆ（”アスキーコード表で対応する文字は　%ｃ　です。￥ｎ”，ｘ）；
}
```

実行結果

変数ｘの内容は１６進表現で　４３　です。

アスキーコード表で対応する文字は　ｃ　です。

変数ｘの内容は１６進表現で　４４　です。
アスキーコード表で対応する文字は　ｄ　です。

ｃｈａｒ型とｉｎｔ型の違いはサイズだけで、フォーマット上の差異はないのでこのようにアスキーコード表に対応する文字がある値の場合ｉｎｔ型の値を%c で文字に置き換えて表示できる。また文字定数と数値定数は、表記上の違いだけでどちらもメモリ上のパターンとして取込まれた結果は整数型である。１バイトの文字定数（半角文字）で表現可能な範囲よりも int 型の変数で扱える範囲の方が広いので、文字定数で記述された値を int 型の変数に代入することができる。

以下のような練習が考えられる。

・変数ｙを用いて、アスキーコード表で５７に対応する文字を表示するプログラムを作成せよ。
・変数ｙを用いて、８進数の１０１と１４１に対応する値を 16 進数と 10 進数で表示するプログラムを作成せよ。
・変数ｘとｙを用いて'R'と'ｒ'に対応する値を 16 進数で表示するプログラムを作成せよ。

プログラム例

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｃｈａｒ　ｙ＝５７；
  ｐｒｉｎｔｆ（”アスキーコード表で 10 進数の%ｄに”，ｙ）；
  ｐｒｉｎｔｆ（”対応する文字は%ｃです。￥ｎ”，ｙ）；
}
```

実行結果

アスキーコード表で１０進数の　57　に対応する文字は　9　です。

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｃｈａｒ　ｙ＝０１０１；
  ｐｒｉｎｔｆ（”８進数の　%ｏ　に対応する 16 進数は　%ｘ　で、”，ｙ，ｙ）；
  ｐｒｉｎｔｆ（”１０進数は　%d　です。￥ｎ”，ｙ）；
  ｙ＝０１４１；
  ｐｒｉｎｔｆ（”８進数の　%ｏ　に対応する 16 進数は　%ｘ　で、”，ｙ，ｙ）；
  ｐｒｉｎｔｆ（”１０進数は　%d　です。￥ｎ”，ｙ）；
}
```

実行結果

８進数の　１０１　に対応する１６進数は　４１　で、１０進数は　６５　です。
８進数の　１４１　に対応する１６進数は　６１　で、１０進数は　９７　です。

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
```

```
    int   x=' R';
    int   y=' r';
    printf (" 文字の%cに対応する１６進数は  %x  です。￥n", x, x);
    printf (" 文字の%cに対応する１６進数は  %x  です。￥n", x, x);
}
```

実行結果

文字のＲに対応する１６進数は　５２　です。

文字のｒに対応する１６進数は　７２　です。

③書式付き入力ｓｃａｎｆ（）

先のプログラムでは5行目にあるように書式付きの入力関数ｓｃａｎｆ（）を用いている。ｓｃａｎｆ（）関数の戻り値が０以上の場合入力項目数を表し、入力に失敗した場合ＥＯＦが戻り値となる。ＥＯＦはｅｎｄ　ｏｆ　ｆｉｌｅの意味で、ヘッダファイルｓｔｄｉｏ．ｈ中に定義されている。通常、標準入力装置はキーボードであるが、Ｃ言語のベースＯＳであるＵＮＩＸでは、標準入力装置からの入力もファイルからの入力と同じように扱えるようになっているため、標準入力関数の入力失敗時の戻り値はＥＯＦとなっている。入力できないということはファイルで言えば、もう取り込む内容がない、ファイルの末尾に達したと同じ意味合いになる。

ｓｃａｎｆ（）関数はｐｒｉｎｔｆ（）関数と同じ様なＳｃａｎｆ変換用文字列リテラルを第１引数として持つが、第２引数以降の記述が異なる。ｐｒｉｎｔｆ（）関数は出力関数で出力されるのは値であるから第２引数以降の引数は式である。それに対してｓｃａｎｆ（）関数は入力されたデータ・値を置く場所を第２引数で指定する。通常、変数はメモリ上の場所に相当する概念であるが、ｓｃａｎｆ（）関数は明示的に値を置く場所のアドレスを指定する必要がある。そのためポインタ変数以外の変数に値を取る場合はは、その変数に対応するアドレスを取り出すアンパサンド演算子・＆と組みあわせて第２引数を記述する。

アンパサンドはアンド記号・＆の事で、アンド記号はアドレス演算子、ビット積演算子、積結合演算子に用いられる。先のプログラムの５行目のように＆キャスト式の形で記述されている場合はアドレス演算子として動作し後に続くキャスト式に対応するアドレス値を取り出す働きをする。対応するアドレスが存在しないレジスタ型等には対応できない。

ｐｒｉｎｔｆ（）関数と異なり、第１引数の文字列リテラル中には、Ｓｃａｎｆ変換を指定する％～以外の文字は記述できない。入力促進用のメッセージ等が必要な場合は、ｐｒｉｎｔｆ（" ｘ＝"）；ｓｃａｎｆ（" %ｄ", ＆ｘ）；のようにｓｃａｎｆ（）関数の前にｐｒｉｎｔｆ（）関数等を用いてメッセージを出力するようにする。先のサンプルプログラムのように同じ行に記述する・しないは動作に無関係である。Ｓｃａｎｆ変換は標準入力装置から渡された文字列の解釈の仕方を指定する。先のサンプルプログラムでは、" %ｄ" で、入力された文字列を整数の１０進数と解釈せよという指定になっている。" %ｉ" は" %ｄ" と同じように

整数値としての解釈を指定しているが”％ｄ”と異なり、入力された文字列の先頭が０なら８進数、０ｘ又は０Ｘなら１６進数として解釈する。

”％ｄ”は１０進数の整数指定で、”％ｏ”は入力文字列の先頭に０があってもなくても８進数の整数指定になり、”％ｘ”は入力文字列の先頭に０ｘや０Ｘがあってもなくても１６進数の整数指定になる。”％ｕ”はｕｎｓｉｇｎｅｄ　ｉｎｔ、符号なし１０数の整数指定になる。”％ｅ”、”％ｆ”、”％ｇ”は浮動小数点の指定を表す。このとき、入力に必ずしも小数点や指数部を含む必要はない。

”％ｓ”は文字・ｃｈａｒ型の入力指定にナルが、ｓｃａｎｆ（）関数は、スペースやタブレーション等は、入力文字列中の各項目の区切りとしてとらえ読み飛ばしてしまうので”％ｓ”では取り込むことができない。”％ｓ”は文字列の指定で、要は入力文字列の末尾に￥０のヌル文字を付加して取り込む。このため受け取る変数は十分な要素数を持ったｃｈａｒ型の配列変数である必要がある。”％ｃ”を用いれば入力文字列を項目に区切らずそのまま取り込めるので、スペース等を取り込みたい場合は”％１ｃ”を用いれば良い。以下にｓｃａｎｆ（）関数の簡単なサンプルプログラムを示す。

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｃｈａｒ　ｃ；
  ｐｒｉｎｔｆ（”￥ｎｃ＝”）；ｓｃａｎｆ（”％ｃ”，＆ｃ）；
  ｐｒｉｎｔｆ（”ｃ＝％ｃ［％２ｘ］￥ｎ”，ｃ，ｃ）；
}
```

実行結果

ｃ＝ｇ　・・・ｇのキーを押してからエンターキーを押した

ｃ＝ｇ［６７］

再度実行

ｃ＝ＡＢ

ｃ＝Ａ［４１］　・・・最初の１文字しかとれていない

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｉｎｔ　ｉ；
  ｐｒｉｎｔｆ（”￥ｎｉ＝”）；ｓｃａｎｆ（”％ｉ”，＆ｉ）；
  ｐｒｉｎｔｆ（”ｉ＝％ｄ［％４ｘ（１６）］［％５ｏ（８）］￥ｎ”，ｉ，ｉ，ｉ）；
}
```

実行結果

ｉ＝１２

```
i＝　１２［　　ｃ（１６）］［　　１４（８）］
```
再度実行
```
ｉ＝０１２　　・・・先頭に０を付けて入力した
i＝　１０［　　ａ（１６）］［　　１２（８）］
```
再度実行
```
ｉ＝０ｘ１２　　・・・先頭に０ｘを付けて入力した
i＝　１８［　１２（１６）］［　　２２（８）］

ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｄｏｕｂｌｅ　ｄ；
  ｐｒｉｎｔｆ（”￥ｎｄ＝”）；ｓｃａｎｆ（”％ｌｆ”、＆ｄ）；
  ｐｒｉｎｔｆ（”ｄ＝％５．２ｆ￥ｎ”，ｄ）；
}
```
実行結果
```
ｄ＝３．１４
ｄ＝　３．１４
```
再度実行
```
ｄ＝７
ｄ＝　７．００　　・・・整数値に見える文字列でも実数型に変換して取り込まれている。
```

入力→処理→出力が基本的なデータの流れで、ここまでの学習で変数への入力としては代入文とｓｃａｎｆ（）関数が、処理には算術式を左辺に持つ代入文が、出力にはｐｒｉｎｔｆ（）関数が利用できるようになった。以下のような動作をするプログラムを作成させてみると良い。

［演習］

［１］キーボードより入力された１文字のアスキーコードを２桁の１６進数で表示するプログラムを作成せよ。

［２］キーボードより円の半径を入力し、その値をもとに面積を表示するプログラムを作成せよ。

＊計算結果の表示は全体７桁、小数点以下２桁とする。

＊計算の有効値の範囲は使用するコンパイラのｄｏｕｂｌｅに依存する。

［３］キーボードから入力されたＡ～Ｚの範囲の大文字１文字をａ～ｚの小文字に変換して表示するプログラムを作成せよ。

＊入力文字がＡ～Ｚの範囲に入っているかのチェックはしない

＊Ａ～Ｚ以外の文字が入力された場合の動作は考えなくても良い

［1］プログラム例

```
#include <stdio.h>
void main(void)
{
  char c;
  pritnf("¥n?:"); scanf("%c", &c);
  printf("文字:%c:アスキーコード:%02x:¥n", c, c);
}
```

［2］プログラム例

```
#include <stdio.h>
#define PI 3.14
void main(void)
{
  double r, s;
  pritnf("¥n半径?:"); scanf("%lf", &r);
  s=r*r*PI;
  printf("半径%fの円の面積:%7.2f¥n", r, s);
}
```

［2］プログラム例　別解

```
#include <stdio.h>
#define PI 3.14
void main(void)
{
  double r;
  printf("¥n半径?:"); scanf("%lf", &r);
  printf("半径%fの円の面積:%7.2f¥n", r, r*r*PI);
}
```

注）printf()関数の第２引数以降の引数は、式であるから、単項式にこだわって変数を増やす必要はない！と言うこともできる。

［3］プログラム例

```
#include <stdio.h>
void main(void)
{
  char c;
  printf("¥n大文字?:"); scanf("%c", &c);
  c=c+('a'-'A');
```

ｐｒｉｎｔｆ（”小文字への変換結果：％ｃ￥ｎ”，ｃ）；

｝

注）ｃ＝ｃ＋（’ａ’－’Ａ’）；とｐｒｉｎｔｆ（”小文字への変換結果：％ｃ￥ｎ”，ｃ）の２行をｐｒｉｎｔｆ（”小文字への変換結果：％ｃ￥ｎ”，ｃ＋（’ａ’－’Ａ’））；の１行にまとめて別解とすることもできる。

（３）基本的な入力関数と出力関数

入力関数は文字入力関数、文字列入力関数、書式入力関数と配列入力関数（ｆｒｅａｄ（）関数）に分類され、出力関数は文字出力関数、文字列出力関数、書式出力関数と配列出力関数（ｆｗｒｉｔｅ（）関数）に分類できる。このうち書式出力関数の内の書式表示関数であるｐｒｉｎｔｆ（）関数と書式入力関数の内の端末書式入力関数であるｓｃａｎｆ（）関数は既に学習した。以下に出力関数と入力関数の一覧を提示する。

出力関数：文字出力関数　：文字表示関数ｐｕｔｃｈａｒ（）
　：ファイル文字出力関数ｆｐｕｔｃ（）
　：ファイル文字出力マクロ関数ｐｕｔｃ（）
　文字列出力関数　：文字列表示関数ｐｕｔｓ（）
　：ファイル文字列出力関数ｆｐｕｔｓ（）
　文字押戻関数　：ｕｎｇｅｔｃ（）
　書式出力関数　：書式表示関数ｐｒｉｎｔｆ（）
　：ファイル書式出力関数ｆｐｒｉｎｔｆ（）
　：文字列編集関数ｓｐｒｉｎｔｆ（）
　可変個引数集出力関数：可変個引数集ファイル出力関数ｖｆｐｒｉｎｔｆ（）
　：可変個引数集表示関数ｖｐｒｉｎｔｆ（）
　：可変個引数集文字列化関数ｖｓｐｒｉｎｔｆ（）
　配列出力関数　：ｆｗｒｉｔｅ（）関数
入力関数：文字入力関数　：端末文字入力関数ｇｅｔｃｈａｒ（）
　：ファイル文字入力関数ｆｇｅｔｃ（）
　：ファイル文字入力マクロ関数ｇｅｔｃ（）
　文字列入力関数　：端末文字列入力関数ｇｅｔｓ（）
　：ファイル文字列入力関数ｆｇｅｔｓ（）
　書式入力関数　：端末書式入力関数ｓｃａｎｆ（）
　：ファイル書式入力関数ｆｓｃａｎｆ（）
　：文字列書式変換関数ｓｓｃａｎｆ（）
　配列入力関数　：ｆｒｅａｄ（）

①文字入力関数＆文字出力関数

［学習項目］ｇｅｔｃｈａｒ（）関数、文字列、ｐｕｔｃｈａｒ（）関数

文字入力関数には端末文字入力関数であるｇｅｔｃｈａｒ（）関数と、ファイル文字入力関数であるｆｇｅｔｃ（）関数と、ファイル文字入力マクロ関数であるｇｅｔｃ（）関数があり、文字出力関数には端末文字出力関数であるｐｕｔｃｈａｒ（）関数と、ファイル文字出力関数であるｆｐｕｔｃ（）関数と、ファイル文字出力マクロ関数であるｐｕｔｃ（）関数がある。Ｃ言語の場合標準入力はいつも利用可能な状態にある特殊なファイルなので、ファイル文字入出力関数やファイル文字入出力マクロ関数を用いて、端末からの文字の入出力が可能である。しかし、ここでは端末文字入出力関数を学習し、制御構文の学習が終わってからファイル入出力関数やファイル入出力マクロ関数の使い方を学習する。

ｐｕｔｃｈａｒ（）関数は、整数型の値を引数に持ち、与えられた整数値に対応する文字を標準出力ストリーム（ｓｔｄｏｕｔ）に出力する。出力した値（文字）そのものを戻り値として返す。もし何らかの理由で書き出しエラーが発生した場合、ＥＯＦの値を戻り値として返す。ｐｕｔｃｈａｒ（）関数を連続して使用した場合、ｐｕｔｃｈａｒ（）関数から１文字出力される毎にカレントの表示位置が１つずつ動いてゆくので表示が重なったりする問題は発生しない。ただしエスケープシーケンスを用いてバックスペース等の制御コードを出力した場合はこの限りではない。以下にｐｕｔｃｈａｒ（）関数の簡単な使用例を示す。

```
#include <stdio.h>
void main(void)
{
  putchar('¥"');
  putchar('h');putchar('e');putchar('l');
  putchar('l');putchar('o');
  putchar('¥"');
  putchar('¥n');
}
```

実行結果

”ｈｅｌｌｏ”

ｇｅｔｃｈａｒ（）関数は、引数を持たず（ｖｏｉｄ）、戻り値として入力ストリーム（ｓｔｄｉｎ）から入力された１文字を返す。標準入力ストリームがファイルの終わりに達した場合、あるいは何らかの理由で、読み込みエラーが発生した場合、ＥＯＦを戻り値として戻す。以下にｇｅｔｃｈａｒ（）関数の簡単な使用例を示す。

```
#include <stdio.h>
void main(void)
{
  int x;
```

```
  x＝ｇｅｔｃｈａｒ（）；
  ｐｒｉｎｔｆ（”ｘ＝％ｃ￥ｎ”，ｘ）；
}
```

実行結果

Ａ　　・・・Ｅｎｔｅｒキーを押すまで次の動作に移らない

ｘ＝Ａ

注）ｇｅｔｃｈａｒ（）関数は、標準入力ストリームから１文字受け取る関数であるが、キーサーチの関数ではないので、多くはＥｎｔｅｒキーが押されるまで次の動作に移らない。

```
#ｉｎｃｌｕｄｅ　＜ｓｔｄｉｏ．ｈ＞
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｐｕｔｃｈａｒ（ｇｅｔｃｈａｒ（））；
  ｐｕｔｃｈａｒ（ｇｅｔｃｈａｒ（））；
}
```

実行結果

ＡＢ

ＡＢ

注）ｇｅｔｃｈａｒ（）関数の戻り値をｐｕｔｃｈａｒ（）関数の引数として渡しているのでＡ、Ｂ、Ｅｎｔｅｒの順にキーを押した場合、キー入力のエコーとｐｕｔｃｈａｒ（）関数の出力、キー入力のエコーとｐｕｔｃｈａｒ（）関数の出力の順番で画面に残りＡＡＢＢとなりそうなものであるが多くの端末で上記のような動作になる。

②文字列入力関数と文字列出力関数

［学習項目］文字列、ｐｕｔｓ（）関数、ｇｅｔｓ（）関数

文字列出力関数には、文字列表示関数であるｐｕｔｓ（）関数とファイル文字列出力関数であるｆｐｕｔｓ（）関数がある。文字列入力関数には端末文字列入力関数であるｇｅｔｓ（）関数とファイル文字列入力関数であるｆｇｅｔｓ（）関数がある。ｆｐｕｔｓ（）関数やｆｇｅｔｓ（）関数を用いて標準入出力ストリームからの入出力も可能であるが、ｆｐｕｔｓ（）関数やｆｇｅｔｓ（）関数は後からファイル操作のところでまとめて学習することにする。

文字列は、定数としては文字列定数・文字列リテラルとして分類されるが文字列定数に直接対応する単純変数は存在しない。文字列定数はｃｈａｒ型の文字データの連なり・集合であるから１文字１文字はｃｈａｒ型の変数に対応する存在である。Ｃ言語において文字列はメモリー上の連続した領域に置かれることが保証されている。このようにメモリ上に連続して置かれているデータを扱うにはデータタイプの如何を問わず配列変数を用いる。文字列はｃｈａｒ型の配列変数に対応する存在である。

配列変数は、単純変数の宣言に続けて必要な変数の個数・連なっているデータの個数を［］

の中に整数定数で記述することにより宣言・確保される。ｃｈａｒ型の連続した領域が６個必要な場合はｃｈａｒ　ｘ［６］；といった具合に記述する。このように宣言するとｘ［０］，ｘ［１］，ｘ［２］，ｘ［３］，ｘ［４］，ｘ［５］という名前で特定可能な６個の変数が連続した領域に確保される。この時［］の中にあって変数名を特定する働きをしている整数値を添え字といい、名前を特定する添え字には整数の評価値を持つ式を埋める事ができる。添え字に対してこの場合ｘは配列変数全体を代表する名前であると言える。［］の前が同じ名前であれば連続したメモリ上に割り振られた同じグループに属していることを示す。実際にｘは配列を宣言した時点で同時に参照のみ可能な変数として先頭の変数ｘ［０］のアドレスを持たされている。このｘのようにアドレスを持つための変数がポインタ型の変数である。ポインタ型の変数を単独で宣言するには、変数の宣言時データタイプの指定に続けて＊を記述する。ｘの宣言を今までの知識をもとに擬似的に書き下してみると、ｃｈａｒ　＊ｘ；でｃｈａｒ型のポインタ変数として宣言しｃｏｎｓｔ　ｘ＝＆ｘ［０］；で先頭のアドレスを設定し読み出し専用に定数化しておく、もっともこのようなｃｏｎｓｔキーワードの使い方は通常はできないのであくまで擬似的な記述である。

文字列を操作する場合ｃｈａｒ型の配列変数が対応する。文字列はその先頭アドレス・ポインタで代表され文字列の末尾は￥０（ヌル文字）で規定される。宣言時のみｃｈａｒ型の配列への文字定数の直接代入が可能である。ｃｈａｒ　ｘ［６］＝”ｈｅｌｌｏ”；といった具合であるこの時ｈｅｌｌｏの５文字に加えて￥０分の１文字分多く領域を確保していることに注意して欲しい。バリエーションとしてｃｈａｒ　ｘ［］＝”ｈｅｌｌｏ”；やｃｈａｒ　＊ｘ＝”ｈｅｌｌｏ”；といった書き方も可能である。ｃｈａｒ　ｘ［］は要素数の数え上げをコンパイラに任せたイメージだが、ｈａｒ　＊ｘは、用意される変数は、定数”ｈｅｌｌｏ”の先頭アドレスを持つためのポインタ変数ｘ１個だけである。ｐｒｉｎｔｆ（）関数の％ｓ指定を用いて文字列を表示するサンプルを以下に提示する。

```
#include <stdio.h>
void main(void)
{
  char x[8]="hello x";
  char y[]="hello y";
  char *z="hello z";
  printf("x[6]=%s¥n", x);
  printf("y[]=%s¥n", y);
  printf("z   =%s¥n", z);
  putchar(x[0]);
  putchar(x[6]);
}
```

実行結果

```
hello x
hello y
hello z
hx
```

puts () 関数は出力文字列を示すポインタ型の引数を持ち標準出力ストリームに指定されたアドレスの内容からヌル文字までの文字列を出力する。この時printf () 関数の%sと異なり最後に改行・¥nを付加して出力する。出力に成功した場合0以上の値を戻り値として戻し、失敗した場合EOFの値を戻す。以下にputs () 関数の簡単な例を示す。

```
#include <stdio.h>
void main(void)
{
  char x[8]="hello x";
  puts("string1");
  puts(x);
  puts(&x[2]);
}
```

実行結果

```
string1     ・・・¥nを付加していないのに改行されている。
hello x
llo x       ・・・x[2]の1から後ろが表示されている。
```

gets関数は、標準入力ストリームからの文字列を改行文字ないしファイルの末尾（EOF）まで読み込む関数で、み込まれた文字列は引数で渡されたアドレスを先頭とするメモリ上に格納される。メモリ上に格納する際、改行文字は取り込まず代わりにヌル文字・空文字が付加される。戻り値はchar型のポインタで文字列を収納したメモリエリアの先頭アドレスが返される。文字列が読み込まれなかった、あるいは途中でエラーが発生した場合NULLポインタが戻される。以下にgets () 関数の簡単な例を示す。

```
#include <stdio.h>
void main(void)
{
  char buff[80];
  gets(buff);
  printf("buff=%s¥n", buff);
  puts(buff);
}
```

実行結果

```
hello world                ・・・キーボードよりの入力
buff=hello world           ・・・printf () 関数の%sによる出力
hello world                ・・・putsによる出力

#include <stdio.h>
void main(void)
{
  char buff[80];
  printf("gets=%s¥n", gets(buff));
  puts(gets(buff));
}
```

実行結果

```
Good morning.
gets=Good morning.     ・・・gets () の戻り値の%sによる出力
Good night.
Good night.                ・・gets () の戻り値のputsによる出力
```

③文字列編集関数sprintf () と文字列書式変換関数sscanf ()

［学習項目］sprintf ()、出力書式、sscanf ()、入力書式

標準入力を既存の文字列に置き換えてとらえると文字列書式変換関数sscanf () 関数になり、標準出力を文字列に対応するchar型の配列変数に置き換えてとらえるとと文字列編集関数sprintf () になる。実質的に文字列編集関数や文字列書式関数は、文字列のパターン変換に用いる。文字列処理関数としてはstring.hに多くの文字列操作用の関数が存在するが、それらはまた順次機会をとらえて学習してゆくとして、ここではstdio.hに含まれ、先に学んだPrintf変換やScanf変換の書式指定が可能なsprintf () 関数とsscanf () 関数の使い方を学習する。

文字列編集関数sprintf () はprintf () 関数の出力先である標準出力が、引数で指定したchar型の配列（厳密にはchar型の配列の先頭アドレス）に入れ替わったものと理解して良い。printf () 関数が第1引数が出力書式文字列で第2引数以降に可変個の引数をもち、第2引数以下の式の評価値を出力書式文字列中の%変換仕様文字で指定される形で埋め込み、評価結果を埋め込んだ文字列を標準出力へ出力するのに対しsprintf () 関数では、printf () 関数の第1引数である出力書式指定文字列が第2引数にきて、第3引数以降に可変個の式がくる。sprintf () 関数の第1引数は、式の評価値を埋め込んだ文字列の収納場所を表すchar型の配列の代表名、すなわちchar型の配列の先頭アドレスを示すポインタである。sprintf () 関数の戻り値は、第1引数で示されるアドレスを先頭として何バイト書き込んだかを示すint型である。戻り値が負の値の場合

何らかのエラーが発生したことになる。以下にｓｐｒｉｎｔｆ（）の基本動作を示す簡単なプログラム例を示す。

```
#include <stdio.h>
void main(void)
{
   char buff[80];
   sprintf(buff,"I am a cat.");
   puts(buff);    ・・・第３引数以降がない例
}
```

実行結果

```
I am a cat.
```

```
void main(void)
{
  char buff[80];
  int x;
  x=sprintf(buff,"I have no name.");
  printf("¥"%s¥":%dバイト¥n", buff, x);
}
```

実行結果

```
"I have no name.":  １５バイト・・・この１５は戻り値の値である。
```

```
void main(void)
{
  char buff[80];
  int x=7;
  sprintf(buff,"I have %d sons.", x);
  puts(buff);
}
```

実行結果

```
I have 7 sons.
```

```
#define PI 3.14
void main(void)
{
```

```
  char  buff[80];
  int  x;
  x=sprintf(buff,"円周率は%5.2fである。", PI);
  puts(buff);
  printf("String has %d bytes.¥n", x);
}
```

実行結果

円周率は　3．14である。

String has 21 bytes.

注）漢字8文字は16バイト%5．2で5バイト長だから計21バイトとなる。

```
void main(void)
{
  char buff[4];
  sprintf(buff,"123456789");
  puts(buff);
}
```

実行結果

123456789

と表示されるかもしれないが、配列のサイズが4バイト分しか確保されていないので動作結果は不定である。Cコンパイラは配列の要素数のチェックはしないのでコンパイルエラーにはならない。C言語における配列の扱いの特徴の確認と文字列は最後に文字列の末尾を示すためにヌル文字･空文字が付加されるので最低1バイト分多く確保しておく必要があることの確認をしておく。

文字列書式変換関数ｓｓｃａｎｆ()は、ｓｃａｎｆ()関数の入力先である標準入力が引数で指定したｃｈａｒ型の配列（厳密にはｃｈａｒ型の配列の先頭アドレス）に入れ替わったものと理解して良い。ｓｃａｎｆ()関数が第1引数が出力書式文字列で第2引数以降に可変個の引数をもち、第2引数以下で指定されたアドレスに位置する変数に%変換仕様文字で指定される形で値を設定するのに対しｓｓｃａｎｆ()関数では、ｓｃａｎｆ()関数の第1引数である出力書式指定文字列が第2引数にきて第3引数以降に可変個のアドレスがくる。ｓｓｃａｎｆ()関数の第1引数は、値を取り出すもとになる文字列を示すｃｈａｒ型の配列の代表名、すなわちｃｈａｒ型の配列の先頭アドレスを示すポインタである。ｓｓｃａｎｆ()関数の戻り値は第1引数で示されるアドレスを先頭とする文字列から第3引数以下で示されるアドレスに位置する変数の何個に値を渡したかを示すｉｎｔ型である。入力に失敗した場合、ＥＯＦが戻り値になる。

以下にｓｓｃａｎｆ（）の基本動作を示す簡単なプログラム例を示す。

```
#include <stdio.h>
void main(void)
{
  int x, y, z;
  sscanf("10 20 30","%d %d %d", &x, &y, &z);
  printf("x=%d, y=%d, z=%d¥n", x, y, z);
}
```

実行結果

```
x= 10, y= 20, z= 30
```

注）ｓｃａｎｆ（）関数のところでは、文章による説明のみであったがこのようにスペース・タブレーション・改行はｓｃａｎｆ（）関数やｓｓｃａｎｆ（）関数に対してセパレータとして作用する。

```
void main(void)
{
    double x;
    int y;
    char z;
    sscanf("3.1¥n9¥tAB","%lf%d%c", &x, &y, &z);
    printf("x=%4.1f, y=%3d, z=%c¥n", x, y, z);
}
```

実行結果

```
x= 3.1, y= 9, z= A
```

注）このように改行やタブレーションもセパレータとして動作する。又％ｃ変換仕様文字で変数ｚには１文字だけしか取り込んでいないことに注意する。

このように変換指定文字の使い方等においてｓｃａｎｆ（）関数とｓｓｃａｎｆ（）関数、ｐｒｉｎｔｆ（）関数とｓｐｒｉｎｔｆ（）の間に差は存在しない。ｓｓｃａｎｆ（）関数の第１引数をｓｔｄｉｎ・標準入力に固定し、第１引数を省略したものがｓｃａｎｆ（）関数、ｓｐｒｉｎｔｆ（）関数の第１引数をｓｔｄｏｕｔ・標準出力に固定し、第１引数を省略したものがｐｒｉｎｔｆ（）であるといえる。

２．制御構造の記述

ここまで文字列を含む基本的なデータタイプと基本入出力関数について学習してきた。ｃｈａｒ型の配列は、文字列として一括して扱える関数が存在するが、一般的に配列は同じ性質を持つデータの集合を扱うのに用いられる。同じ性質を持つデータに対する操作が同一であっても、反復文を用いなければデータの個数だけ同じ記述を繰り返すことになる。又関数の戻り値においては多くの関数がエラー時の戻り値を規定している。標準入出力に対する入出力やバッファ・配列に対する入出力の場合入力対象が存在しない、あるいは出力ストリームが確保できないといった事態は比較的まれである。ファイルにおいてはまずファイルを使う前に読み出したいファイルが存在するのか、出力ファイルの領域は確保できるか等の確認を行ってから、本来の入出力処理の記述にかかる必要がある。そういった確認を行うためには判断制御の記述ができる必要がある。ここでは判断記述、反復文の記述方法について学習する。

（１）判断制御の記述

関数の戻り値が正常値か否かの確認等、式の値を判断し、次に続くブロックの内容を実行するか否かによってプログラムを分岐させる判断制御を行うのがｉｆ文である。ｉｆ文は、ｉｆ（制御式）真文あるいはｉｆ（制御式）真文ｅｌｓｅ偽文の二つのスタイルがあり、真文及び偽文は１つだけのステートメントで構成される時は、ブロックの始まりと終わりを示す中括弧を持たない。制御式は通常論理式であるが、論理式以外の式の記述も許す。制御式の評価結果が非０の値（真）ならば真文を実行し、０（偽）ならば真文を実行せず、もし偽文が存在する場合偽文を実行する。

①制御式

［学習項目］ｉｆ文、制御式、比較演算子、論理結合演算子

ｉｆ文は後に続く制御式と真文によって構成される。真文はｉｆ文自身も含む通常のステートメントである。真文を構成するステートメントが１文のみの時は、ブロックを構成する中括弧が省略可能である。真文・偽文がない時は、{} で中身のないブロックを指定するか、; を打ち忘れないようにする。打ち忘れると、プログラマの意識上ではｉｆ文と考えて、ｉｆ文の次の動作を指定するステートメントが、真文あるいは偽文として解釈され実行されることになる。

制御式は、その評価結果が真（非０）であるか偽（０）であるかだけがｉｆ文の動作に影響を与える。制御式の記述には任意の演算子を用いた記述が可能である。四則演算等の基本的な演算子は既に学習した。ここでは制御式の記述と関連づけて比較演算子と論理結合演算子を学習する。

比較演算子には不等演算子と等非等演算子がある。不等演算子は、値の大小を比較する演算子である。２項演算子で不等演算子の前後に式を持ち、各式の評価値の関係が不等演算子で記述した関係になっていれば真を、なっていなければ偽を評価結果とする。不等演算子には小なり演算子、大なり演算子、以下演算子と以上演算子の４つがある。小なり演算子は＜の記号で、大なり演算子は＞の記号で表現される。以下演算子と以上演算子はそれぞれ小なり（＜）ある

いは大なり（＞）記号と等号（＝）の組み合わせで表現される。それぞれ以下・以上の関係を示すが、二つの記号で１つの演算子を表すこと、以下演算子は＜＝（小なりイコール）、以上演算子は＞＝（大なりイコール）でしか表現できないことに注意する必要がある。等非等演算子は、値が等しいか等しくないかを比較する演算子である。２項演算子で等非等演算子の前後に式を持ち、各式の評価値の関係が等非等演算子で記述した関係になっていれば真を、なっていなければ偽を評価結果とする。等非等演算子には等価演算子と非等価演算子の２つがある。等価演算子は＝＝（イコールイコール）の記号で、非等価演算子は！＝（感嘆符イコール）の記号で表現される。このとき＝＝の最初のイコールは代入操作を意味しないが、非等価演算子の！＝は、以下演算子や以上演算子と同じように、この順番でしか表現できない。

論理結合演算子には積結合演算子と和結合演算子がある。論理結合演算子は前後に式を持つ２項演算子で各式の評価結果の真（非０）と偽（０）をもとに論理演算を行い真（非０）ないし偽（０）を評価結果とする。積結合演算子は＆＆（アンド・アンド）で表現され、和結合演算子は｜｜（二重縦線）で表現される。積結合演算子は式１＆＆式２の記述になる２項演算子で式１と式２の評価結果がともに真の時のみ、評価結果が真となる。和結合演算子は式１｜｜式２の記述になる２項演算子で式１と式２の評価結果がともに偽の時のみ、評価結果が偽となる。積結合演算子を表す＆＆は、必ず＆を２個連続して記述する。＆だけの場合ビット単位の論理積を求める演算子を表し意味が変わってしまう。和結合演算子を表す｜｜は、必ず｜を２個連続して記述する。｜だけの場合ビット単位の論理和を求める演算子を表し意味が変わってしまう。以下に論理結合演算子の真理値表及び真文のみで構成される簡単なプログラムの例を示す。

式１	式２	式１＆＆式２	式１｜｜式２
偽	偽	偽	偽
偽	真	偽	真
真	偽	偽	真
真	真	真	真

```
#include <stdio.h>
#define K 50
void main(void)
{
  int x;
  printf("¥nx="); scanf("%d", &x);
  if (x>K)
    printf("%d is larger than %d.¥n", x, K);
  if (x<K)
    printf("%d is less than %d.¥n", x, K);
}
```

実行結果

```
x=55
  55 is larger than 50.
x=35
  35 is less than 50.
```

```
void main(void)
{
  char x;
  printf("¥nx="); scanf("%c", &x);
  if ('A'<=x) && (x<='Z'))
    printf("%c is upper character.¥n", x);
  if ('a'<=x) && (x<='z'))
    printf("%c is lower character.¥n", x);
}
```

実行結果

```
x=d
d is lower character.
x=H
H is upper character.
x=9
```

・・・2つのｉｆ文のどちらの条件にも不一致

```
void main(void)
{
  char x;
  printf("¥nx=");scanf("%c", &x);
  if ((('A'<=x) && (x<='Z'))
          || (('a'<=x) && (x<='z')))
    printf("%c is an alphabet.¥n", x);
}
```

実行結果

```
x=q
q is an alphabet.
x=R
R is an alphabet.
x=8
```

②else文

[学習項目] else文、if～elseの入れ子構造

else文は単独で用いる事はなく、if文に偽文を持たせるために使用する。記述はif(制御式)真文else偽文となる。制御文の記述は真文のみのif文と同じである。制御式の評価値が真の時は真文を実行し、偽の時は偽文を実行する。if文と同じ様に真文ないし偽文にif文やif～else文を含めることもできる。以下に簡単なif～else文の例を示す。

```
#include <stdio.h>
#define L_A 80
#define L_B 60
#define L_C 40
void main(void)
{
  int x;
  printf("¥nx(100～0)=");scanf("%d", &x);
  if ((x<0) || (100<x)) {
    printf("ERROR¥n");
  } else {
    if (x>L_A) {
      printf("A class¥n");
        } else {
```

```
      if (x>L_B) {
        printf ("B  class¥n");
      } else {
            if (x>L_C) {
        printf ("C  class¥n");
        } else {
        printf ("D  class¥n");
        }
      }
    }
  }
}
```

実行結果

```
x (0～100) =105
ERROR?
x (0～100) =95
A  class
x (0～100) =45
C  class
x (0～100) =15
D  class
```

③switch文とif文

［学習項目］switch～case文、caseラベル

前述の例のように、ある入力値に対して幅を持つ範囲で対応を場合分けする場合はif～else文を用いる。しかし入力値が整数値である場合、あるいは整数値に置き換え可能な場合、例えば、入力値として1～12の範囲の整数値を受け取り、入力値4に対してウルトラマン、入力値6に対してドラエモン、入力8に対してサザエさん、入力10に対してコナン、入力値2に対してニュース、入力値12に対して見たくない、その他の範囲内の値（1，3，5，7，9，11）に対しては受信不能のメッセージを出力するプログラムを考えてみる。まず、if～elseを用いて記述してみる。

```
void  main (void)
{
  int  x;
  printf ("¥nx (1～12) ="); scanf ("%d", &x);
  if ((x<1) || (12<x)) {
```

```
    printf (" ERROR¥n");
  } else {
    if (x==12)
      printf (" 見たくない¥n");
        else
      if (x==10)
            printf (" コナン¥n");
      else
        if (x==8)
          printf (" サザエさん¥n");
        else
          if (x==6)
            printf (" ドラエモン¥n");
          else
            if (x==4)
              printf (" ウルトラマン¥n");
            else
              if (x==2)
                printf (" ニュース¥n");
              else
                printf (" 受信不能¥n");
  }
}
```

この例では、入力値の範囲チェック以外は、真文・偽文ともに1ステートメントで構成されいるので、まだif～elseの入れ子構造に混乱の生じる余地が少ないが（混乱が生じないようにインデントを付けて記述するのであるが)、真文や偽文がブロック・中括弧による記述になってくると、入れ子構造の記述が煩雑になってくる。これをswitch～case文を用いて記述してみる。

```
void  main (void)
{
  int  x;
  printf (" ¥nx (1～12) ="); scanf (" %d", &x);
  if ((x<1) || (12<x)) {
    printf (" ERROR¥n");
  } else {
    switch (x) {
```

```
    case  12:
      printf ("見たくない¥n");  break;
        case  10:
          printf ("コナン¥n");      break;
    case  8:
      printf ("サザエさん¥n");  break;
    case  6:
      printf ("ドラエモン¥n");  break;
    case  4:
      printf ("ウルトラマン¥n"); break;
    case  2:
      printf ("ニュース¥n");    break;
    default:
      printf ("受信不能¥n");
    }
  }
}
```

このように分岐条件が整数値の場合、ｓｗｉｔｃｈ～ｃａｓｅ文を用いた方がはるかにわかり易い記述になる。ｃａｓｅ　に続く整数値の後ろはコロンである。ｃａｓｅ　整数値：は正確にはラベルの一種でｃａｓｅラベルといいＣ言語でもラベルを示すのに幾つかのアセンブラと同じように：（コロン）を用いる。各ケースに対応する動作の記述の最後にはｂｒｅａｋ文を付けるべきである。ｉｆ～ｅｌｓｅでは真文・偽文の実行は排他的で真文を実行後偽文を実行することはあり得ないが、ｓｗｉｔｃｈ～ｃａｓｅ文ではｂｒｅａｋ；がないと次のｃａｓｅラベルの値の後に続く動作に移ってしまう。このサンプルではｄｅｆａｕｌｔラベルに対応するｂｒｅａｋ文がないが、これはｄｅｆａｕｌｔラベルがｓｗｉｔｃｈブロックの最後のラベルでその後ろにラベルが並んでいないためである。動作の性質はｉｆ～ｅｌｓｅに似ているがラベルで、分岐先を記述しているため若干注意が必要であるが、その他の一括した記述がｄｅｆａｕｌｔで可能な事等がやや魅力的であろう。

（２）反復文

反復文は繰り返し文ともいい、ｗｈｉｌｅ文、ｄｏ文、ｆｏｒ文の３種類がある。全ての反復文は制御式を持ち、制御式の評価結果が０（偽）になるまでループ本体を繰り返し実行する。ループ本体が１ステートメントの場合は、中括弧で囲んでブロック化する必要はない。ループ本体にステートメントを全く含まない場合は反復文に続けて {} か；だけを記述する。反復文は３種類もなくてもこのうちどれか１つの反復文とｉｆ文、ｂｒｅａｋ文、ｃｏｎｔｉｎｕｅ文があれば構造的にアルゴリズムを記述することが可能である。まずｗｈｉｌｅ文について学習し、その後前判定反復文であるｗｈｉｌｅ文に対して、後判定文であるｄｏ～ｗｈｉｌｅ文

についても学習する。

①前判定反復文

［学習項目］前判定反復文、ｗｈｉｌｅ文、合計の計算、カウンタ、キャスティング

反復文をステートメントとして持つのが高水準言語の特徴であるが、基本的な機能しか持たないＣＰＵでは、制御用の命令としてｉｆ文とｇｏｔｏ文に相当するステートメントしか持たないものも多い。高水準言語で記述されたステートメントは最終的には、ＣＰＵに直接理解可能な命令・機械語に置き換えて実行される。ｗｈｉｌｅ文等の制御構文も最終的にはｉｆ文とｇｏｔｏ文の組み合わせで表現し直される。この時反復文の制御式はｉｆ文の制御式に渡されると考えて良い。制御式がそのまま渡されるときはｉｆ文の真文中にｇｏｔｏ文が含まれる。制御式はループを繰り返す条件の記述であるから、ｇｏｔｏ文の行き先を示すラベルはループの先頭になり、ｉｆ文はループの末尾に位置することになる。これが後判定の繰り返しである。ループの末尾で次のループを実行するかどうかを判定するのであるからループは最低一回は実行される。これに対して反復文の制御式の真偽を反転して判別する形でｉｆ文に渡すとそのｉｆ文はループの終了条件・脱出条件を持つことになる。このようなｉｆ文はループの末尾にループの先頭のラベルを持つｇｏｔｏ文を置いておけば、ｉｆ文自体はループ中の任意に場所に置くことができる。当然ループの先頭に置いても良い。ループの先頭にループの終了条件を持つｉｆ文を置いたのが前判定の繰り返しである。ループを実行する前に脱出条件を吟味するから一回もループが実行されない場合もあり得る。機械語では前判定の時は脱出条件、後判定の時は繰り返し条件を記述することになる。Ｃ言語では後判定の時も、前判定の時も条件式には繰り返し条件を記述する。前判定か後判定かは、用いるステートメントで決定される。ｗｈｉｌｅ文を用いれば前判定の反復文、ｄｏ～ｗｈｉｌｅを用いれば後判定の反復文になる。

ｗｈｉｌｅ文は、ｗｈｉｌｅ（制御式）{ループの本体} となる、ループブロックを構成するステートメント数には制限はなく、ループ文中にｉｆ文や他の反復文やｗｈｉｌｅ文を含めるこができる。ｗｈｉｌｅ文は前判定の反復文であるからループブロックの反復回数は０回以上である。以下にｗｈｉｌｅ文の簡単な例を示す。

```
#include <stdio.h>
void main(void)
{
  int x=0;
  while (x>=0) {
    printf("x="); scanf("%d", &x);
  }
}
```

これは、変数ｘの内容が０以上の間、変数ｘへの標準入力からの入力を繰り返すプログラムである。これはｉｆ文とｂｒｅａｋ文を用いて以下の様に書き直す事もできる。

```
void  main (void)
{
  int  x;
  while (1) {
    printf (" x="); scanf (" %d", &x);
    if (x<0) break;
  }
}
```

while文の制御式に定数を持たせることもできる。while（非0）はwhile（真）であるから無限ループの記述になる。無限ループはこのようにbreak文を持つif文と組みあわせて用いる。ちなみにwhile（0）はwhile（偽）であるからループブロックは1回も実行されない。

この例では先ほどの例と異なりint　x＝0；の初期化がなされていないことに注意して欲しい。先ほどの例では制御式がx＞＝0でこのwhile文にたどり着く前に変数xが0以上の確定した値を持っていないとループブロックに実行に取りかかれない、しかもC言語の変数は宣言しただけでは、その持つ値は不定である。

ここで標準入力から、負の値が入力されるまで0以上の値の入力を繰り返し、入力された0以上の値の合計を計算し計算結果を表示するプログラムを考えさせる。最初の例と後の例で合計を計算するs＝s＋x；はループブロック中どこに入れれば良いのか、xの初期値等に注意を払う必要はないのか等について注意を促す。後の例ではif文後にs＝s＋x；を以下のように追加することになる。

```
void  main (void)
{
  int  x, s;
  s=0;
  while (1) {
    printf (" x="); scanf (" %d", &x);
    if (x<0) break;
    s+=x;／*  s=s+x  *／
  }
  printf (" Total=%d¥n", s);
}
```

実行結果例

```
x＝10
x＝21
```

ｘ＝３２
ｘ＝－９
Ｔｏｔａｌ＝　６３

ｓ＋＝ｘはＣ言語特有の記述で、式の右辺を構成する式が？２項式でかつ評価値を受け取る変数が式の最初の項を構成している場合、このような省略した記述が可能になる。一種の略記と捉えることも可能であるが、この記述の方が変数ｓにｘの内容を足し込むという内容に即した記述である。また、変数ｓは宣言しただけではその値は不定で、０クリアを忘れると宣言時に持っている不定値も足し込んでしまうことになる。先の例で同様の動作をさせるにはどのような記述になるのであろうか、以下に例を示す。

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｉｎｔ　ｘ＝０；
  ｉｎｔ　ｓ＝０；
  ｗｈｉｌｅ（ｘ＞＝０）{
    ｓ＋＝ｘ；
    ｐｒｉｎｔｆ（”ｘ＝”）；ｓｃａｎｆ（”％ｄ”，＆ｘ）；
  }
  ｐｒｉｎｔｆ（”Ｔｏｔａｌ＝％ｄ￥ｎ”，ｓ）；
}
```

今度はｓｃａｎｆ（）関数の前にｓ＋＝ｘ；を記述する必要がある。ｓｃａｎｆ（）の後だとループブロック脱出用の－９を足し込んでしまうことになる。まずｘの値が０でｓ＝０＋０でｓ＝０、次に１０が入力されｓ＝０＋１０でｓ＝１０、次に２１が入力されｓ＝１０＋２１でｓ＝３１、次に３２が入力されｓ＝３１＋３２でｓ＝６３となる、さらに－９が入力され－９＞＝０の評価値は偽であるから次のループには入らずｐｒｉｎｔｆ（”Ｔｏｔａｌ＝％ｄ￥ｎ”，ｓ）；が実行される。実行結果は同じであるが、ｘの初期値が足し込まれている。ループの中に入るには０以上の値であれば何でもいいのだが、ｘの値は内部で足し込まれるのでここでは０以外の値は設定し得ないことになる。ここでは注意深くプログラムをトレースする練習も兼ねて動作を確認させておく。

次に平均を求めるプログラムを考えさせてみる、ｉｆ（）ｂｒｅａｋを含むｗｈｉｌｅ文の使い方の練習と１を足し込んでいくという発想でカウンタを思いつかせる練習である。以下にプログラム例を示す。

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｉｎｔ　ｘ，ｓ；
  ｉｎｔ　ｉ；
  ｓ＝０；ｉ＝０；
```

```
  while (1) {
    printf (" x="); scanf (" %d", &x);
    if (x<0) break;
    i+=1;
    s+=x;／*  s=s+x  *／
  }
  printf (" Total=%d¥t", s);
  printf (" average=%d¥n", s／i);
}
```

実行結果例

```
x=10
x=21
x=32
x=-9
Total=  63          average=  21
```

i=i+1;のように1を足し込む動作を記述しておけばこのステートメントは1回実行される毎に変数iの内容が1つづつ増加してゆくので、逆に変数iの値を読めばこのステートメントが何回実行されたか判断できる。このように実行された回数を持つ変数をカウンタ・カウンタ用変数・カウンタ変数と呼びi=i+1やi+=1のようなステートメントを記述する事をカウンタを置くと言ったりする。カウンタ変数iの使い方と合計を取る変数sのはどちらも足し込む動作の対象であるのでゼロクリアが必要である。

カウンタのように1を足し込む動作や逆に1を引いてゆく動作はループの繰り返し回数等の確認によく用いる。多くのCPUは機械語レベルで1を足し込む（1増やす）インクリメント命令や1を引く（1減らす）デクリメント命令を持っている。インクリメント命令やデクリメント命令を持たない高水準言語も多いが、Cはインクリメント演算子（++）とデクリメント演算子（--）を持つ。i+=1はインクリメント演算子を用いて++i又はi++と書き直せる。インクリメント演算子を左側に置く場合はインクリメントしてから評価、右に置く場合は評価してからインクリメントとなり、インクリメント演算子を右に書くか左に書くかによって評価値に影響がでるが、ここではインクリメント演算子を含む式が単独で記述されており評価対象になっていないのでどちらでも良い。

又この実行結果例では、偶然平均値が割り切れる値になっているが、割り切れない値になることの方が多い。例えば%6.1fなどとすると全体で6桁、小数点以下1桁の表示を指定できるが、表示対象となる式はdouble型ないしfloat型のデータタイプである必要がある。このような場合C言語では型の明示的な変換が可能である。通常int型の変数sとint型の変数iの演算結果はint型であるが、式の前に（double）と書けば評価値を

ｄｏｕｂｌｅ型に変換してくれる。（データタイプ）で指定されたデータタイプに変換する事が可能である。この機能を明示的なキャスティング機能という。インクリメント演算子とキャスティングを含むように上記の例を書き直してみる。

```
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｉｎｔ　ｘ、ｓ；
  ｉｎｔ　ｉ；
  ｓ＝０；ｉ＝０；
  ｗｈｉｌｅ（１）{
    ｐｒｉｎｔｆ（”ｘ＝”）；ｓｃａｎｆ（”％ｄ”，＆ｘ）；
    ｉｆ（ｘ＜０）ｂｒｅａｋ；
    ＋＋ｉ；／＊評価対象になっていないので、ここはｉ＋＋でも同じ動作になる＊／
    ｓ＋＝ｘ；／＊　ｓ＝ｓ＋ｘ　＊／
  }
  ｐｒｉｎｔｆ（”Ｔｏｔａｌ＝％ｄ￥ｔ”，ｓ）；
  ｐｒｉｎｔｆ（”ａｖｅｒａｇｅ＝％６．１ｆ￥ｎ”，
                                    （ｄｏｕｂｌｅ）ｓ／（ｄｏｕｂｌｅ）ｉ）
；
}
```

実行結果例

```
ｘ＝１２
ｘ＝２３
ｘ＝３５
ｘ＝－９
Ｔｏｔａｌ＝　７０　　ａｖｅｒａｇｅ＝　２３．３
```

②後判定反復文

後判定反復文はｄｏ文を用いて記述する。ｄｏ｛ループ内容｝ｗｈｉｌｅ（制御式）；の形で記述する。ｗｈｉｌｅ文の場合ｗｈｉｌｅ（制御式）の後にセミコロンは置かないが、ｄｏ｛｝ｗｈｉｌｅ（制御式）；でセミコロンを置く必要がある。以下にｄｏ文の簡単な例を示す。

```
＃ｉｎｃｌｕｄｅ　＜ｓｔｄｉｏ．ｈ＞
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
{
  ｉｎｔ　ｘ；
  ｄｏ｛
    ｐｒｉｎｔｆ（”ｘ＝”）；ｓｃａｎｆ（”％ｄ”，＆ｘ）；
  ｝ｗｈｉｌｅ（ｘ＞＝０）；
```

```
}
```

ｗｈｉｌｅ文の最初の例と同じ動作をするプログラムであるが、ｄｏ文は最低一回はループを実行するので、最初にループの中に入るための設定は不必要になっている。

③ｃｏｎｔｉｎｕｅ、ｂｒｅａｋ

既に確認したように、ｂｒｅａｋ文はｗｈｉｌｅ文やｓｗｉｔｃｈ文の｛｝ブロックから強制的に脱出する働きをする。これはｄｏ文においても同様である。ｓｗｉｔｃｈ文のｃａｓｅラベルに対応するブロックの最後には単独で、ｗｈｉｌｅ文やｄｏ文のループブロック中にはｉｆ文の真文として登場する。反復文と組みあわせられた場合ｂｒｅａｋ文はそこで繰り返しを終了させる。例えば次のような例を考えてみる。

```
ｗｈｉｌｅ（1）｛
  ｐｒｉｎｔｆ（”ｓｔｅｐ1￥ｎ”）；
  ｂｒｅａｋ；
  ｐｒｉｎｔｆ（”ｓｔｅｐ2￥ｎ”）；
｝
```

ｗｈｉｌｅ（1）は無限ループであるが、ｂｒｅａｋ文が単独で記述されているので、ｓｔｅｐ1を1回表示しただけでこのループは終了してしまう。上記のようなループでｂｒｅａｋ文の代わりにｃｏｎｔｉｎｕｅ文を置くとどうなるのであろうか。

```
ｗｈｉｌｅ（1）｛
  ｐｒｉｎｔｆ（”ｓｔｅｐ1￥ｎ”）；
  ｃｏｎｔｉｕｅ；
  ｐｒｉｎｔｆ（”ｓｔｅｐ2￥ｎ”）；
｝
```

これはｓｔｅｐ1を何行も表示し続ける無限ループになってしまう。（注、無限ループに入っても標準出力への出力を含んでいるので、多くのＯＳでは＾Ｃ・コントロールＣ等でキーボードから強制中断可能である。）ｃｏｎｔｉｕｅ文は名前の通り、ループをｃｏｎｔｉｎｕｅ（続ける）のである。ただし何もせずに続けたのでは意味がない、ｃｏｎｔｉｎｕｅ文からループブロックの最後・｝（閉じ中括弧）までを省略し次のループに取りかかるのである。ｃｏｎｔｉｎｕｅ文も多くの場合ｂｒｅａｋ文と同様にｉｆ文の真文ないし偽文としてループ注に含まれる。ｃｏｎｔｉｎｕｅ文を用いると以下のように、－１００から＋１００の範囲の値の入力を受け付け正の値だけの合計を計算するプログラムが記述できる。

```
＃ｉｎｃｌｕｄｅ　＜ｓｔｄｉｏ．ｈ＞
ｖｏｉｄ　ｍａｉｎ（ｖｏｉｄ）
｛
  ｉｎｔ　ｘ、ｓ；
  ｓ＝0；
  ｗｈｉｌｅ（1）｛
```

```
    printf (" x="); scanf (" %d", &x);
    if ((x<-100) || (100<x)) break;
    if (x<0) continue;
    s+=x;／*  s=s+x  *／
  }
  printf (" Total=%d¥n", s);
}
```

実行結果例

```
x=10
x=-21
x=32
x=-999
Total=  42
```

注）この例は、xの値が負の時は以下の処理を省略するという記述であるが。xの値が正の時だけ足し込む対象とすると捉えると、if（x<0）continue；s+=x；の2行がif（x>0）s+=x；の1行ですむ。

（3）所定回反復文

先にwhile文の例でカウンタを用いてループの回数を数え上げるプログラムを示したが、実際のプログラムでは繰り返しの最大回数が決まっている反復が多い。このような場合while文中のカウンタの値をif文で判断しbreakすれば良いのであるが、若干煩雑であるし、こういったタイプの反復を数え上げ型の反復文・所定回反復文と呼ぶ。数え上げ型の反復なら数え上げ型の反復として対応する反復文があった方が可読性の面からいっても有利である。数え上げ型の反復を示すのがfor文である。高水準言語の中にはBASIC言語のように反復文はfor文しか持たないものも存在する。for文はwhile文と比較すると複合的な反復文である。for文はwhile文と同じようにforに続けて（）丸括弧で囲まれたブロックを持つが、while文は（）・丸括弧の中に制御式・継続条件式しか持たないのに対してfor文の（）中には初期設定式、継続条件式、再設定式の3つの内容をこの順番にセミコロンで区切って記述する。このうち継続条件式・制御式はwhile文の制御式と同じである。すなわちfor（初期設定式；継続条件式；再設定式）{ループ本体}のスタイルで記述される。break文、continue文に対する反応は、while文と同じである。初期設定式は最初の1回目のループを実行する前に1回だけ実行され、継続条件式は毎回ループの実行にかかる前に吟味され、再設定式は毎回ループを実行する前に実行される。

①数え上げ型の繰り返し

［学習項目］最大繰り返し回数の決まっている繰り返しの実現方法

```
#define  MAX  3
void  main（void）
```

```
{
  int   i=0;
  while (i<MAX) {
    printf (" i=%d¥n", i);
    ++i;
  }
}
```

実行結果例

```
i=0
i=1
i=2
```

```
#define  MAX  3
void  main (void)
{
  int   i;／＊i=0の初期化はここでは必要がない＊／
  for (i=0; i<MAX; ++i) {
    printf (" i=%d¥n", i);
  }
}
```

実行結果例

```
i=0
i=1
i=2
```

この２つのプログラムはループを３回繰り返す動作をｗｈｉｌｅ文とｆｏｒ文を用いて記述したものである。繰り返しの制御式こそｉ＜ＭＡＸと同じであるが、ｗｈｉｌｅ文ではｗｈｉｌｅループへ入る前に必要であった、制御変数ｉの初期化がｆｏｒ文の初期設定式に、ループ中のカウンタのインクリメントがｆｏｒ文の再設定式に含まれている。このｉ＝０の初期設定と＋＋ｉと制御式が一体になって反復構造を構成している。ｆｏｒ文ではこの３つの式がｆｏｒに続く（）・丸括弧中に含まれ一体感を強く感じさせる。ｆｏｒ文の方が可読性が高いはずである。ここで以下のような練習をしても良い。

＜練習＞

［１］３個の整数を標準入力から受け取り、受け取った３個の値の合計を表示するプログラムを作成せよ。

［２］５個の整数を標準入力から受け取り、受け取った５個の値の合計と平均を表示するプログラムを作成せよ。

［1］の解答例

```
#define MAX 3
void main(void)
{
  int i;
  int x, s;／* 変数xとsは同じタイプの変数 *／
  for (i=0, s=0; i<MAX; ++i) {
    printf ("x=");
    scanf ("%d", &x);
    s+=x;
  }
  printf ("sum=%d¥n", s);
}
```

［2］の解答例

```
#define MAX 5
void main(void)
{
  int i;
  int x, s;／* 変数xとsは同じタイプの変数 *／
  for (i=0, s=0; i<MAX; ++i) {
    printf ("x=");
    scanf ("%d", &x);
    s+=x;
  }
  printf ("sum=%d¥n", s);
  printf ("ave=%f¥n", double s／MAX);
}
```