プログラミング言語 C--

Ver. 3.0.0

徳山工業高等専門学校
情報電子工学科

目 次

第1章　はじめに

C--言語は C 言語に似た小さなシステム記述用言語です。徳山高専教育用 PC(TaC) のシステム記述用言語として開発されました。C--言語は次に挙げる項目を満たすことを目標に設計されています。

「学習が容易な言語であること」 C--言語は C 言語をお手本にしていますが、C 言語の難しい部分を取り去り簡単に理解できるようになっています。まず、無くても我慢できそうな文法は、思い切り良く省略しています。例えば、C 言語では多次元配列の形式にいくつかのレパートリーがありました。しかし、Java 言語にはレパートリーはありません。C--言語は Java 言語に倣い多次元配列のレパートリーを認めません。

また、C 言語の混乱を招きそうな文法仕様を取り入れないように注意しています。例えば、C 言語の配列は関数に渡されるとポインタとして扱われます。つまり、関数に渡すと型が変化してしまいます。このような仕様は、初心者が言語を学習する場合に混乱を招きます。C--言語は Java の参照の考えかたを取り入れ、配列は一貫して参照型として取り扱われます。

その他にも、C 言語の難しい文法を取り去る工夫をしてあります。Java 言語と似た仕様にすることにより、Java 言語でプログラミングの入門をした人が学習しやすくなっています。

「実用的なシステム記述言語であること」 C--言語は TaC のシステム記述言語として、TaC の OS や C--コンパイラを記述することを目標にしています。そのため、無闇に文法を簡単化すること無く、実用的に使用するために必要な文法は残してあります。例えば、制御文は `if`、`while`、`for`、`do-while`、`return`、`break`、`continue` 等が一通り準備されています。

また、なるべく効率の良いオブジェクトコードを出力する努力をしています。

「TaC 上で実行可能なこと」 最終的に TaC 上でセルフ開発環境を構築することを目標としています。C--コンパイラは、セルフ開発環境の中核になるコンポーネントです。そこで、C--コンパイラは TaC の限られた主記憶 (64kiB) で実行できるようにメモリを節約するような設計がされています。コンパイラのプログラムが小さいこともそうですが、単一の名前表で変数、関数、構造体を管理する等してデータ構造も小さくするようにしています。

「コンパイラを教材として使用できること」 C--コンパイラは、高専や大学の学生が、コンパイラの教材として使用することを想定して開発されました。そのため、コンパイラがコンパクトに記述でき、コンパイラのソースコードを学生が読めることも目標になっています。

2016年2月現在のC--コンパイラはC言語で5,000行程度で記述されています。5,000行の内訳は、おおよそ、字句解析部が500行、構文・意味解析部が1,200行、中間コード生成部が800行、機械語コード生成部が900行、名前表管理部が200行、構文木管理部が300行、構文木の最適化部が600行です。少し根気が必要ですが、各モジュールを順に読んでいくことができます。

第2章　C--言語開発環境のインストール

C--言語を体験するために、自分のパソコンでC--コンパイラを使用できるようにしましょう。

2.1　ユーティリティのインストール

TaC用のプログラムを作成するために、まず、C--コンパイラ用ユーティリティをインストールする必要があります。ソースコードは https://github.com/tctsigemura/Util--/ から入手します。

ダウンロードした配布物を解凍し「Util--解説書」(Util--/doc/umm.pdf）の手順に従いインストールします。as--、ld--、objbin--、objexe--、size--の五つのプログラムが/usr/local/bin にインストールされます。

2.2　コンパイラのインストール

C--コンパイラのソースコードは https://github.com/tctsigemura/C--/ から入手します。ダウンロードした配布物を解凍し以下の順にインストールします。

2.2.1　コンパイラ本体のインストール

C--/src ディレクトリに移動し以下のように操作します。

```
$ make
cc -std=c99 -Wall  -DDATE="\"‘date‘\"" -DVER=\"3.0.0\" ...
...
$ sudo make install
Password:
install -d -m 755 /usr/local/bin
install -m 755 c-- /usr/local/bin
install -m 755 vm-c-- /usr/local/bin
install -m 755 c-c-- /usr/local/bin
```

以上で、c--、vm-c--、c-c--の三つのプログラムが作成され/usr/local/bin にインストールされました。

- c--は、TaC 用の本物のコンパイラです。TaC のアセンブリ言語を出力します。
- vm-c--は、コンパイラの勉強をしたい人のために、中間言語 (61 ページ) をよく反映した、仮想スタックマシンのニーモニックを出力して見せるコンパイラです。
- c-c--は、C--言語を普通のパソコンやマイコンで試してみたい人のために、C--言語を C 言語に変換するトランスレータです。変換された C 言語プログラムは普通のパソコンで実行できるはずです。

2.2.2　ヘッダファイルのインストール

C--言語プログラム用のヘッダファイルをインストールします。C--/include ディレクトリに移動し以下のように操作します。/usr/local/cmmInclude にインストールされます。

```
$ sudo make install
Password:
install -d -m 755 /usr/local/cmmInclude
rm -f /usr/local/cmmInclude/*.hmm
install -m 644 *.hmm /usr/local/cmmInclude
```

2.2.3　ライブラリのインストール

C--/lib ディレクトリに移動しまずライブラリをコンパイルします。以下のように操作します。

```
$ make
as-- crt0.s
cc -E -std=c99 -nostdinc  -I/usr/local/cmmInclude - < ctype.cmm |
  c--  > ctype.s
as-- ctype.s
...
ld-- libtac.o crt0.o ctype.o stdio.o stdlib.o string.o ...
rm stdio.s string.s stdlib.s ctype.s
```

以上で、libtac.o が作成されました。これが、TacOS のアプリケーションプログラムにリンクされるライブラリです。次に、インストールします。操作は次の通りです。

```
$ sudo make install
Password:
install -d -m 755 /usr/local/cmmLib
install -m 644 libtac.o /usr/local/cmmLib
install -m 644 cfunc.hmm /usr/local/cmmLib
install -m 644 wrapper.c /usr/local/cmmLib
```

/usr/local/cmmLib ディレクトリに、いくつかのファイルがインストールされました。libtac.o は C--言語で記述した TacOS 用のライブラリ関数です。cfunc.hmm は C--プログラムがパソコンで実行される際に C--言語ライブラリ関数の代用として使用される C 言語ライブラリ関数を定義します。wrapper.c は C--プログラムがパソコンで実行される際に使用される C 言語で記述された C--ライブラリ関数です。cfunc.hmm による置き換えができない関数が記述してあります。

2.3 サンプルプログラムのコンパイル

C--/samples/hello ディレクトリに移動します。hello.cmm プログラムが準備してあります。これをコンパイルしてみましょう。なお、C--/samples/c--ディレクトリには、C--言語で記述した C--コンパイラが置いてあります。後で、こちらのコンパイルにも挑戦してください。

2.3.1 コンパイル

このディレクトリで make コマンドを実行します。下に示す実行例のように hello.cmm が自動的にコンパイルされ、hello.exe、hello、hello.c、hello.s、hello.v の五つのファイルができます。

実行例 2 行は hello.cmm から hello.c を作る手順を示しています。実行例 3 行は hello.c から hello を作る手順を示しています。実行例 4 行は hello.cmm から hello.v を作る手順を示しています。実行例 5 行は hello.cmm から hello.s を作る手順を示しています。実行例 6〜8 行は hello.s から hello.exe を作る手順を示しています。実行例を参考に、自分の目的に必要な操作法方法を確認してください。

```
1: $ make
2: cc -E -DC -std=c99 -nostdinc -I/usr/local/cmmInclude
   -I/usr/local/cmmLib - <  hello.cmm | c-c-- > hello.c
3: cc -o hello -Wno-parentheses-equality hello.c
   /usr/local/cmmLib/wrapper.c
4: cc -E  -std=c99 -nostdinc -I/usr/local/cmmInclude
   -I/usr/local/cmmLib - <  hello.cmm | vm-c-- > hello.v
5: cc -E  -std=c99 -nostdinc -I/usr/local/cmmInclude
   -I/usr/local/cmmLib - <  hello.cmm | c-- > hello.s
6: as-- hello.s
7: ld-- mod.o /usr/local/cmmLib/libtac.o hello.o > hello.sym
8: objexe-- hello.exe mod.o 600 | sort --key=1 > hello.map
```

2.3.2 関連ファイル

上の実行例で、使用したり作成されたりしたファイルの主なものを紹介します。

- hello.cmm は、C--言語のもっとも簡単なサンプルプログラムです。
  画面に、hello,world と表示します。

```
//
// hello.cmm : C--のサンプルプログラム
//
#include <stdio.hmm>

public int main() {
  printf("hello,world\n");
  return 0;
}
```

- hello.exe は、hello.cmm をコンパイルした TacOS 用の実行形式ファイルです。TaC の実機で実行できます。
- hello は、hello.cmm をコンパイルしたパソコン用の実行形式のファイルです。次のようにパソコンで実行できます。

```
$ ./hello
hello,world
```

- hello.c は、hello.cmm を c-c-- で C 言語に変換したプログラムです。上記の hello は hello.c を C 言語コンパイラでコンパイルしたものです。

```
#include <stdio.h>
#define _cmm_str_L0 "hello,world\n"
int main(){
printf(_cmm_str_L0);
}
```

- hello.s は、hello.cmm を c--で TaC のアセンブリ言語に変換したプログラムです。上記の hello.exe ファイルは、このファイルから C--コンパイラ用ユーティリティ (Util--) を用いて作成されました。

```
_stdin  WS      1
_stdout WS      1
_stderr WS      1
.L1     STRING  "hello,world\n"
_main   PUSH    FP
        LD      FP,SP
        CALL    __stkChk
        LD      G0,#.L1
        PUSH    G0
        CALL    _printf
        ADD     SP,#2
        LD      G0,#0
        POP     FP
        RET
```

- hello.v は、hello.cmm を vm-c--により仮想スタックマシンのニーモニックに変換したプログラムです。中間言語の詳細は、61 ページで紹介します。

```
_stdin
        WS      1
_stdout
        WS      1
_stderr
        WS      1
.L1
        STRING  "hello,world\n"
_main
        ENTRY   0
        LDC     .L1
        ARG
        CALLF   1,_printf
        POP
        LDC     0
        MREG
        RET
```

- Makefile は、コンパイル手順を記述したファイルです。これまでの実行例で、make コマンドで自動的にコンパイルができたのは、このファイルのおかげです。C--プログラムは、コンパイル前にプリプロセッサで処理する必要があります。コンパイルする度にプリプロセッサやコンパイラを順に手動で起動するのは面倒なので、Makefile を準備した方が良いと思います。

  C--/samples/hello ディレクトリには、先程の実行例で使用された Makefile の他に、実際に使用する場合を想定した MacOS、UNIX 用の Makefile.unix と、TacOS 用の Makefile.tac が準備してあります。以下に後の二つのファイルの内容を掲載します。これらは、OSX 10.10 で動作するように調整してあります。これを参考に、自分の目的と環境に合った Makefile を作ってください。

```
#
# Makefile.unix : C--言語から MacOS や UNIX の実行形式に変換する手順
#
INCDIR=/usr/local/cmmInclude
LIBDIR=/usr/local/cmmLib
CFLAGS+=-Wno-parentheses-equality
CPP=cc -E -DC -std=c99 -nostdinc -I${INCDIR} -I${LIBDIR} - <
.SUFFIXES: .o .cmm .c
.cmm.c:
        ${CPP} $*.cmm | c-c-- > $*.c
OBJS=hello.o
hello: ${OBJS}
        cc -o hello ${CFLAGS} ${OBJS} ${LIBDIR}/wrapper.c
clean:
        rm -f hello ${OBJS}
```

```
#
# Makefile.tac : C-- 言語から TacOS の実行形式に変換する手順
#
INCDIR=/usr/local/cmmInclude
LIBDIR=/usr/local/cmmLib
CFLAGS+=-Wno-parentheses-equality
CPP=cc -E -std=c99 -nostdinc -I${INCDIR} - <
.SUFFIXES: .o .cmm .s
.cmm.s:
        ${CPP} $*.cmm | c-- > $*.s
.s.o:
        as-- $*.s
OBJS=hello.o
hello.exe: ${OBJS}
        ld-- mod.o ${LIBDIR}/libtac.o ${OBJS} > hello.sym
        objexe-- hello.exe mod.o 600 | sort --key=1 > hello.map
clean:
        rm -f hello.exe mod.o ${OBJS} *.sym *.lst *.map
```

第3章　C--言語の仕様

システム記述言語として実績のある C 言語を参考に C--言語は設計されました。しかし、C 言語は設計が古く分かりにくい仕様が多い言語でもあります。そこで、設計が割と新しい Java 言語を参考に C 言語の問題点を避けるようにしました。最終的に C--言語は、**「Java 言語の特徴を取り入れた簡易 C 言語」** になりました。

3.1　コメント

C--言語では、2 種類のコメントを使用できます。一つは /* 〜 */ 形式のコメント、もう一つは // 〜 行末 形式のコメントです。以下に例を示します。コメントは、空白を挿入できるところならどこにでも書くことができます。

```
/*
 *  コメントの例を示すプログラム
 */
#include <stdio.hmm>              // printf を使用するために必要

public int main( /* 引数なし */ ) {
  printf("%d\n",10);
  return 0;                       // main は int 型なので
}
```

3.2　プリプロセッサ

C--言語の仕様ではありませんが、通常 C--コンパイラは C 言語用のプリプロセッサと組み合わせて使用します。前の章に Makefile の例がありましたが、プリプロセッサとコンパイラをパイプで接続して使用します。プリプロセッサのお陰で、#define、#include 等のディレクティブを使用することができます。

C--コンパイラはプリプロセッサが処理した結果を受け取ります。受け取った入力の、ヘッダファイル内部に相当する部分にエラーを見つけた場合でも、正しくエラー場所をレポートできます。また、C 言語を出力するトランスレータとして動作する場合は、出力し

た C 言語プログラム中に適切な#include ディレクティブを出力したり、ヘッダファイル内部を省略して出力する等の処理をしています[1]。

3.3 データ型

C--言語のデータ型は、Java 言語と同様に基本型と参照型に大別できます。基本型には、int 型、 char 型、 boolean 型の 3 種類があります。参照型には、配列型と構造体型があります。参照型は Java 言語の参照型と良く似た型です。

3.3.1 基本型

int 型は整数値の表現に使用します。char 型は文字の表現に使用します。boolean 型は true と false のどちらかの値を取る論理型です。Java 言語の boolean 型と同じものです。違う型の間での代入はできません。また、自動的な型変換も行いません[2]。

int 型

int 型は 16bit 符号付き 2 の補数表現 2 進数です。(トランスレータ版では C 言語の int 型に置き換えますので、C コンパイラに依存します。) -32768 から 32767 までの範囲の数値を表現することができます。int 型変数は次のように宣言します。

```
// int 型変数を宣言した例
int a;                          // int 型のグローバル変数
int b = 10;                     // 初期化もできる
public int main() {
  int c;                        // ローカル変数
  int d = 20;                   // 初期化もできる
  return 0;
}
```

int 型の定数は、上のプログラム例にあるように 「10」、「20」と書きます。また、16 進数、8 進数で書くこともできます。上の例と同じ値を、16 進数では「0xa」、「0x14」のように、8 進数では「012」、「024」のように書きます。int 型は、四則演算、ビット毎の論理演算、シフト演算、比較演算等の計算に使用できます。

[1] 前章の hello.c の例を見てください。
[2] この仕様は厳しすぎました。将来、一部の自動的な型変換は許可します。

char 型

char 型は文字を格納するデータ型です。内部表現は 8bit の ASCII コードですが整数型の一種ではなく、int 型との自動的な型変換や四則演算等ができません。char 型を用いてできる計算は同値比較だけです[3]。「chr 演算子」、「ord 演算子」を用いた明示的な型変換により、char 型の文字と int 型の ASCII コードの間で相互変換ができます。

```
// char 型変数を宣言した例
char a;                         // char 型のグローバル変数
char b = 'A';                   // 初期化もできる
public int main() {
  char c;                       // ローカル変数
  char d = 'D';                 // 初期化もできる
  int  i = ord(b);              // char 型から ASCII コードに変換
  c = chr(i);                   // ASCII コードから char 型に変換
  return 0;
}
```

char 型の定数は、上のプログラム例にあるように 「'A'」、「'D'」と書きます。制御文字を表現するために表 3.1 のエスケープ文字が用意されています[4]。

表 3.1: エスケープ文字

エスケープ文字	意味
\n	改行
\r	復帰
\t	水平タブ
\x16 進数	文字コードを 16 進数で直接指定
\8 進数	文字コードを 8 進数で直接指定
\文字	印刷可能な文字を指定 (\' や \\等)

boolean 型

C 言語では int 型で論理値を表現しました。そのため、条件式を書かなければならないところに間違って代入式を書いたミスを発見できず苦労することがよくありました。C--言語では論理型 boolean を導入したので、このようなミスをコンパイラが発見できます。

[3] この仕様も厳しすぎました。将来、大小比較も許可する予定です。
[4] 正確には、コンパイラではなくアセンブラが、エスケープ文字の処理をしています。

```
// C 言語でよくある条件式と代入式の書き間違え
if (a=1) {
  ...
}
```

boolean 型は論理演算と同値比較演算のオペランドになることができます。boolean 型の定数値は、true、false と書き表します[5]。boolean 型も int 型と互換性がありません。「bool 演算子」、「ord 演算子」による明示的な型変換を用いると int 型との変換が可能です。

次に boolean 型の変数を宣言して使用する例を示します。

```
// boolean 型の使用例
boolean b = true;     // true は定数

public int main() {
  boolean c = x==10;  // 比較演算の結果は boolean 型
  b = b && c;         // 論理演算の結果も boolean 型
  if (b) {            // 論理値なので条件として使用できる
    b = false;        // false も定数
    ...
  }
  int i = ord(b);     // boolean 型から内部表現への変換
  b = bool(i);        // 内部表現から boolean 型への変換
  ...
}
```

3.3.2 参照型

参照型は C 言語のポインターに似た型です。Java 言語の参照型と非常に良く似ています。参照型には配列型と構造体型があります。参照型の値はインスタンスのアドレスです。参照型の特別な値として何も指していない状態を表す null があります。null の値はアドレスの 0 です。ここでは、配列、多次元配列、文字列、 void 配列、構造体について解説します。

[5] true の内部表現は 1、false の内部表現は 0 です。

配列

C--言語でも配列を使用することができます。配列は「型名 []」と宣言します。例えば、int 型や char 型の配列 (参照変数) は次のように宣言します。

```
// 配列の参照変数を宣言した例
int[] a;
char[] b;
```

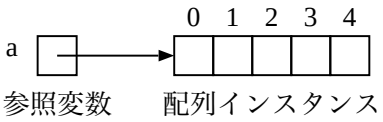

図 3.1: 配列の構造

図 3.1 のように、C--言語の配列は参照変数と配列インスタンスの組合せによって実現します。上の宣言では、参照変数が生成されるだけです。配列として使用するためには次の例のように「array」や「malloc」を用いて配列インスタンスを割り付ける必要があります。「array」は、配列インスタンスを静的に割り付けます。「array」は、関数の外で宣言される配列だけで使用できます。「malloc」は、配列インスタンスを実行時に動的にヒープ領域に割り付けます。「malloc」で割り付けた領域は、使用後「free」によって解放する必要があります。

```
// 配列インスタンスを割り付ける例
int[] a, b = array(10);       // 要素数 10 の int 配列領域を割り付ける
public int main() {
  a = malloc(sizeof(int)*5); // 要素数 5 の int 配列領域を割り付ける
  ...
  free(a);                    // malloc した領域は忘れず解放する
  return 0;
}
```

配列は次のようなプログラムでアクセスできます。添字は 0 から始まります。

```
// 配列をアクセスする例
b[9] = 1;
a[0] = b[9] + 1;
```

多次元配列

多次元配列は配列の配列として表現されます。図 3.2 に示すように、1 次元配列の参照を要素とした配列を使うと 2 次元配列になります。下に、2 次元配列を使用するプログラム例を示します。プログラム中の a2 は、malloc によって、図 3.2 のようなデータ構造に作り上げられます。「malloc」を使用する場合は、プログラムが複雑になってしまいますが、長方形ではない配列も実現できます。プログラム中 b2 は、「array」を使用して多次元配列に必要な配列インスタンスを割り付けた例です。このように「array」を使用すると、多次元配列に必要な複雑なデータ構造を簡単に割り付けることができます。

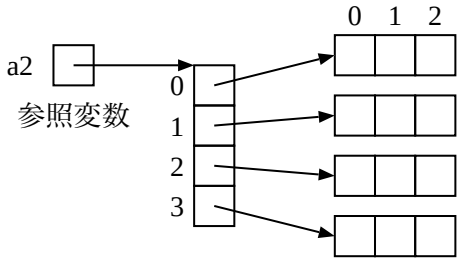

図 3.2: 2 次元配列の構造

```
// 2 次元配列の例
int[][] a2, b2 = array(4,3);        // 2 次元配列の領域を割り付ける
public int main() {
  a2 = malloc(sizeof(void[])*4);    // 参照の配列を割り付ける
  for (int i=0; i<4; i=i+1)
    a2[i] = malloc(sizeof(int)*3); // int 型の 1 次元配列を割り付ける
  ...
  for (int i=0; i<4; i=i+1)
    free(a2[i]);                    // int 型の 1 次元配列を解放
  free(a2);                         // 参照の配列を解放
  return 0;
}
```

次のプログラムは多次元配列をアクセスする例です。このプログラムは、図 3.2 の右下の要素 (最後の要素) に 1 を代入したあと、それを使用して b2 の要素の値を決めます。

```
  a2[3][2] = 1;
  b2[3][2] = a2[3][2] + 1;
```

文字列

C 言語同様に、文字列は char 型の配列として表現されます。文字列は、文字コード 0x00 の文字で終端されます。次のプログラム例のように文字列定数を用いることができ、また、文字列の終端を '\0' との比較で判断できます。

```
// 文字列の使用例
char[] str = "0123456789";
void putstr() {
  for (int i=0; str[i]!='\0'; i=i+1)  // 文字列の終端まで
    putchar(str[i]);                  // 一文字づつ出力
}
```

void 配列

void 型の配列は、どんな参照型とも互換性がある特別な型になります。C--言語は型チェックが厳しいので、異なる型の間での代入はできません。しかし、次のプログラム例のように void 型配列を用いることにより、異なる参照型の間で代入ができます。また、malloc 関数や free 関数は、void 型配列を使って実現されています。

```
// 型変換の例
struct XYZ { int x,y,z; };
XYZ p = { 1, 2, 3 };
void putXYZ() {
  void[] tmp = p;             // tmp にはどんな参照も代入可能
  int[]  xyz = tmp;           // tmp からはどんな参照へも代入可能
  for (int i=0; i<3; i=i+1)
    printf("%d\n",xyz[i]);    // 構造体を配列として無理矢理アクセス
}
```

構造体

C--言語の構造体は Java 言語のクラスからメソッドを取り除いたものに良く似ています。構造体は次のプログラム例のように宣言します。この例は重連結リストを構成するための Node 型を宣言しています。C 言語の構造体宣言に良く似ていますが、構造体名が型名になる点と、構造体が参照型である点が異なります。

```
struct Node {
  Nodo next;    // 自身と同じ型を参照
  Nodo prev;    // 自身と同じ型を参照
  int  val;     // ノードのデータ
};
```

Node 型の変数を宣言して使用する例を下に示します。構造体名が型名になるので struct を書かずに変数宣言します。構造体メンバは、「参照 . メンバ名」形式で参照します。new の代わりに malloc を使用する他は Java 言語と良く似た書き方になります。

```
public int main() {
  Node start;                          // 重連結リストのルート
  start = malloc(sizeof(Node));        // 番兵をリストに投げ込む
  start.next = start;                  // 番兵を初期化する
  start.prev = start;
  start.val  = 0;
  ...
}
```

だまし型

内容不明のまま参照型を宣言できます。他の言語で記述された関数を呼び出す場合など、単に参照型として扱うだけで十分な場合があります。typedef の後に名前を書きます。以下に使用例を示します。

```
typedef FILE;   // FILE 型を定義する

void f() {
  FILE fp = fopen(...); // C 言語の関数を呼び出す
  fprintf(fp,...);      // C 言語の関数を呼び出す
  ...
}
```

3.4 関数

C--言語でも、C 言語同様に関数を宣言して使用することができます。C 言語と比較して C--言語は厳格さを求めています。つまり、先に宣言された関数しか呼び出すことができませんし、引数の個数や型が完全に一致しないとコンパイル時エラーになります。ライブラリ関数を使用する場合は、呼び出す前に必ずプロトタイプ宣言をするか、適切なヘッダファイルをインクルードする必要があります。また、関数の型を省略することができません。値を返さない関数は void、整数を返す関数は int、論理値を返す関数は boolean 等と明示する必要があります。

C 言語や Java 言語と同様に関数の仮引数は、関数の自動変数と同じように使用できます。可変個引数の関数を宣言することもできます。可変個引数関数の仮引数は「...」と書きます。可変個引数関数の内部では、47 ページで説明する_args 関数を使用して引数をアクセスします[6]。次にプログラム例を示します。

```
#include <stdio.hmm>        // printf のプロトタイプ宣言が含まれる
int f() {                   // 引数の無い関数
  return 1;                 // void 型以外の関数は必ず return が必要
}

void g(int x) {
  x = x * x;                // 仮引数は変数のように使用できる
  printf("%04x\n", x);      // プロトタイプ宣言が必要
}

public int main() {
  int x = f();              // 関数の呼び出し
  g(x);                     // 引数の型と個数が一致する必要がある
  return 0;                 // void 型以外の関数は必ず return が必要
}
```

3.5 interrupt 関数

interrupt 関数は、OS カーネル等の割込みハンドラを C--言語で記述するために用意しました。コンパイラに-K オプションを与えないと使用できません。

interrupt 関数は CPU のコンテキスト (フラグ、レジスタ等) を全く破壊しません。関数の入口でコンテキストをスタックに保存し、出口で復旧します。割込みにより起動される

[6] C 言語へのトランスレータ版では、可変個引数関数を定義することができません。C 言語の可変個引数関数 (print 等) を呼び出すことだけできます。

関数なので、プログラムから呼び出すことはできません。仮引数を宣言することもできません。次の例のように関数型の代わりに interrupt と書きます。例中の main 関数のように、addrof 演算子 (26 ページ参照) や_iToA 関数 (45 ページ参照) を使用して、割込みベクタに interrupt 関数を登録します。

```
interrupt timerHdr() {          // タイマー割込みハンドラのつもり
  ...
}
public int main() {
  int[] vect = _iToA(0xffe0);  // vect は割込みベクタの配列
  vect[4] = addrof(timerHdr);  // vect[4] はタイマー割込みのベクタ
  ...
}
```

3.6 変数

C--言語の変数は静的な変数と自動変数の 2 種類です。関数の外部で宣言した変数は全て静的な大域変数になります。逆に、関数の内部で宣言した変数は全てブロック内でローカルな自動変数になります。関数の内部では、どこでも変数宣言が可能です。ローカル変数の有効範囲はブロックの終わりまでです。同じ名前の変数があった場合はローカル変数が優先されます。ローカル変数同士の名前の重複は認めません (Java 言語と同じ規則、C 言語と異なる)。次にプログラム例を示します。

```
#include <stdio.hmm>
int   n = 10;                  // 静的な変数
int[] a = array(10);           // 静的な配列
public int main() {
  int i;                       // 関数内ローカル変数
  for (i=0;i<n;i=i+1) {        // 大域変数 n のこと
    int j = i * i;             // ブロック内ローカル変数
    printf("%d\n",j);
    int n = j * j;             // どこでも変数宣言可能
    printf("%d %d\n",j,n);     // ローカルな n のこと
  }
  printf("%d\n",n);            // 大域変数 n のこと
  return 0;
}
```

3.7 変数の初期化

基本型の変数は、いつでも宣言と同時に初期化することができます。参照型の変数は静的に割り付けられる場合だけ初期化できます。構造体内部に入れ子になった参照型は null で初期化することしかできません。ただし名前表のようなデータ構造を作るために、文字列での初期化だけは可能にしてあります。

```
int     n = 10;                                  // 基本型変数の初期化
int[]   a = { 1, 2, 3 };                         // 基本型配列の初期化
int[][] b = {{1,2,0},{1,2,3,0}};                 // いびつな配列の初期化
struct List { int  val; List next; };            // 構造体の宣言
List    r = { 1, null };                         // 構造体変数の初期化
struct NameEntry { char[] name, int val; };
NameEntry[] weekTable = {                        // 名前表を作成する例
  {"Sun", 1}, {"Mon", 2}, {"Tue", 3}
};
public int main() {
  int i = 10;                                    // 自動変数の初期化
  return 0;
}
```

3.8 public 修飾子

関数、大域変数を他のコンパイル単位から参照できるようにします。public 修飾子の付いていない関数や大域変数は他のコンパイル単位からは見えないので、重複を心配しないで自由に名前を付けることができます。main 関数はスタートアップルーチンから呼び出されるので、必ず public 修飾をしなければなりません。

```
int        n = 10;            // 同じ .cmm ファイル内だけで参照可
public int m = 20;            // 他の .cmm ファイルからも参照可

void f() { ... }              // 同じ .cmm ファイル内だけで参照可
public void g() { ... }       // 他の .cmm ファイルからも参照可

public void printf(char[] s, ...);  // ライブラリ関数は public

public int main() {          // main は必ず public
  f();
  g();
  printf("\n");
  return 0;
}
```

3.9 演算子

C--言語には C 言語をお手本に一通りの演算子が準備されています。しかし、コンパイラを小さくする目的で、レパートリーの多い代入演算子、前置後置等の組合せが複雑なインクリメント演算子とデクリメント演算子を省略しました。

3.9.1 代入演算

C 言語や Java 言語には、たくさんの種類の代入演算子があり便利に使用できました。C--言語では、コンパイラをコンパクトに実装するために代入演算子を「 =」の 1 種類だけにしました。C 言語や Java 言語同様、代入演算の結果は代入した値になります。

C--言語では、代入演算子の左辺と右辺が厳密に同じ型でなければなりませんこれは、コンパイル時になるべく多くのバグを発見するための仕様です。参照型の場合も型が厳密に一致している必要があります。ただし、void[] だけ例外的にどの参照型とも代入可能です。自動的な型変換はありません。

```
int     a;
boolean b = true;
char    c;
struct X { int r; };
struct Y { int r; };
a = 10;             // 同じ型なので代入可能
c = a;              // （エラー）型が異なるので代入できない
a = b;              // （エラー）型が異なるので代入できない
int i = a = 9;      // 代入演算 (a=9) の結果は代入した値 (9)
                    // 代入演算の結果 (9) を i に代入する
X x = { 1 };
Y y;
y = x;              // （エラー）型が異なるので代入できない
void[] p = x;       // void[] にはどんな参照型も代入可能
y = p;              // void[] はどんな参照型にも代入可能
```

3.9.2 数値演算

int 型データの計算に、2 項演算子の「+」(和)、「-」(差)、「*」(積)、「/」(商)、「%」(余) が使用できます。その他に、単項演算子「+」、「-」が使用できます。演算子の優先順位は数学と同じです。計算 (数値演算) をして、計算結果を変数に代入 (代入演算) する例を次に示します。

```
x = -10 + 3 * 2;
```

3.9.3 比較演算

(1) 整数型の大小比較と同値の判定、(2) 参照型、文字型、論理型の同値の判定ができます。大小比較の演算子は、「>」(より大きい)、「>=」(以上)、「<」(未満)、「<=」(以下) の 4 種類です。同値を判定する演算子は、「==」(等しい)、「!=」(異なる) の 2 種類です。比較演算の結果は論理型です。比較演算の結果を論理型変数に代入することができます。論理型は if 文や while 文などの条件に使用できます。次に、比較演算の例を示します。

```
int x = 11;
boolean b;
b = x > 10;                 // 整数の大小比較
if (b==false) { ... }       // 論理型の同値判定
```

3.9.4 論理演算

論理型のデータを対象にした演算です。演算結果も論理型になります。単項演算子「!」(否定)、2 項演算子「&&」(論理積)、「||」(論理和) が使用できます。次に、論理演算の例を示します。論理型変数 b に比較結果を求めた後で、b の否定を if 文の条件に使用しています。

```
int x = 11;
boolean b;
b = 10 <= x && x <= 20;  // (10<=x) と (x<=20) の論理積を b に代入
if (!b) { ... }          // 論理値の否定
```

3.9.5 ビット毎の論理演算

整数値を対象にした演算です。演算結果も整数値になります。単項演算子「~」(全ビットを反転)、2 項演算子「&」(ビット毎の論理積)、「|」(ビット毎の論理和)、「^」(ビット毎の排他的論理和) が使用できます。次に、ビット毎の論理演算の例を示します。マスクを使用して、変数 x の下位 8 ビットを取り出して表示します。printf の括弧内で下位 8 ビットを取り出すためにビット毎の論理演算をしています。

```
int x   = 0xabcd;
int msk = 0x00ff;
printf("%x", x & msk);
```

3.9.6 シフト演算

整数値を対象にした演算です。演算結果も整数値になります。2 項演算子「>>」(右算術シフト)、「<<」(左算術シフト) が使用できます。算術シフトしかありません。次に、シフト演算の例を示します。シフト演算とマスクを使用して、変数 x の値の上位 8 ビットを取り出して表示します。算術シフトですから、マスクを忘れないように注意する必要があります。printf の括弧内で上位 8 ビットを取り出す計算をしています。

```
int x   = 0xabcd;
printf("%x", x >> 8 & 0x00ff);
```

3.9.7 参照演算

配列と構造体をアクセスするための「[添字式]」や「.」は、参照を対象にする演算子と考えることができます。「[添字式]」演算子は、配列参照と添字式から配列要素の値を求めます。「.」演算子は、構造体参照とメンバ名からメンバの値を求めます。配列の配列である多次元配列のアクセスは、「[添字式]」演算子により取り出した配列要素値が配列参照なので、更に「[添字式]」演算子により次の配列要素を取り出すと考えます。実際、C--コンパイラの内部でもそのように考えて扱っています。次に多次元配列や構造体を使用したプログラムの例を示します。「3.3.2 参照型」に示したプログラム例も参考にしてください。

```
// 多次元配列は配列参照の配列と考える
int[][] a = {{1,2},{3,4}};  // 2次元配列を作る
void f() {
  int[] b = a[0];           // 2次元配列の要素は1次元配列
  int   c = b[1];           // 1次元配列の要素は int 型
  int   d = a[0][1];        // c と d は同じ結果になる
}

// 構造体リスト例
struct List {               // リスト構造のノード型
  List next;                // 次のノードの参照
  int  val;                 // ノードの値
};
List a;                     // リストのルートを作る

void g() {
  a = malloc(sizeof(List));       // リストの先頭ノードを作る
  a.val  = 1;
  a.next = malloc(sizeof(List));  // リストの2番目ノードを作る
  a.next.val  = 2;
  a.next.next = null;             // 2番目ノードは参照の参照
}
```

3.9.8 sizeof 演算

変数のサイズを知るための演算子です。malloc で領域を割り付けるとき使用します。「sizeof(型)」のように使用します。型が基本型の場合は「変数の領域サイズ」、構造体の場合は「インスタンスの領域サイズ」をバイト単位で返します。型が配列型の場合は、何型の配列かとは関係なく「参照の領域サイズ（アドレスのバイト数)」を返します。通常、参照の領域サイズは「sizeof(void[])」と書きます。配列インスタンスのサイズが必要なときは、sizeof の値に配列の要素数を掛け算して求めます。以下に使用例を示します。

```
// sizeof 演算子の使用例
int a = sizeof(int);                    // int のサイズ（TaC 版で 2）
int b = sizeof(char);                   // char のサイズ（TaC 版で 1）
int c = sizeof(boolean);                // boolean のサイズ（TaC 版で 1）

int d = sizeof(void[]);                 // 参照のサイズ（TaC 版で 2）
struct X { int x; int y; };
int e = sizeof(X);                      // 構造体 X のサイズ（TaC 版で 4）

int[] f = malloc(sizeof(int)*3);        // 大きさ 3 の int 配列を割当
X[] g = malloc(sizeof(void[])*3);       // 大きさ 3 の参照配列を準備
g[0] = malloc(sizeof(X));               // 構造体インスタンスを割当
g[1] = malloc(sizeof(X));
g[2] = malloc(sizeof(X));
```

3.9.9 addrof 演算

関数や大域変数のアドレスを知るための演算子です。「addrof(大域名)」のように使用し整数型の値を返します。interrupt 関数を割込みベクタに登録したりする目的で使用します。配列や構造体の要素や、関数のローカル変数のアドレスを求めることはできません。

3.9.10 ord 演算

char 型、boolean 型の値を int 型に変換します。char 型の場合は文字の ASCII コード、boolean 型の場合は true=1、false=0 となります。

3.9.11 chr 演算

int 型の ASCII コードから、char 型の文字に変換します。

3.9.12 bool 演算

int 型の 1、0 から、boolean 型の論理値に変換します。
以下に、ord、chr、bool 演算子の使用例を示します。

```
// ord(), chr(), bool() 演算子の使用例
int i = 0x41;         // 'A' の ASCII コード
char c = chr(i);      // c に、文字'A' が代入される
c = chr(ord(c)+1);    // c に、文字'B' が代入される
i = ord(true);        // i は 1 になる
boolean b = bool(1); // b は true になる
```

3.9.13 カンマ演算

複数の式を接続して文法上一つの式にします。例えば、式が一つしか書けない for 文の再初期化部分に二つの式を書くために使用できます。カンマ演算子は、最も優先順位の低い演算子です。

```
// カンマ式を使用した例
for (i=0;i<0;i=i+1,j=j-1) { ... } // 再初期化に二つの式を書いた
```

3.9.14 演算子のまとめ

関数の呼び出しも厳密には「(引数リスト)」演算子と考えることができますが、関数の呼び出しかたは 19 ページで説明したので省略します。その他に、演算の順序を明確にするための「(式)」も含めて、演算子の優先順位を表 3.2 にまとめます。表の上の方に書いてある演算子が優先順位の高い演算子です。同じ高さにある演算子同士は同じ優先順位になります。計算は優先順位の高いものから順に行われます．例えば,「*」は「+」よりも優先順位が高いので先に計算されます．優先順位が同じ場合は結合規則の欄で示した順に計算されます。

3.10 文

関数の内部に記述されるもので、変数宣言以外の記述を文と呼びます。文は機械語に変換されて実行されます。文には、空文、式文、ブロック、制御文 (if 文や for 文) 等があります。C--言語の制御文には switch 文がありませんが C 言語にある他の制御文は一通り揃っています。

表 3.2: 演算子の優先順位

演算子	結合規則
sizeof()、addrof()、ord()、chr()、bool()、関数呼出、()、[]、.	左から右
+(単項演算子)、-(単項演算子)、!、~	右から左
*、/、%	左から右
+、-	左から右
<<、>>	左から右
>、>=、<、<=	左から右
==、!=	左から右
&	左から右
^	左から右
|	左から右
&&	左から右
||	左から右
=	右から左
,	左から右

3.10.1 空文

単独の「;」を空文と呼び、文法上、一つの文として扱います。本文のない for 文等で形式的な本文として使用します。次に、空文を用いる例を示します。

```
for (i=2; i<n; i=i*i)  // 必要なことはこの 1 行で全部記述できた
  ;                    // 空文
```

3.10.2 式文

式の後ろに「;」を付けたものを式文と呼び、文法上、一つの文として扱います。C--言語の文法に代入文はありませんが、代入式に「;」を付けた「式文」が同じ役割に使用されます。

式 ;

3.10.3 ブロック

「{」と「 }」で括って複数の文をグループ化し、文法上、一つの文にします。if 文や while 文の「本文」は、文法的には一つの文でなければなりません。複数の文を「本文」

として実行させたい場合はブロックにします。また、ブロックはローカル変数の有効範囲を決定します。ブロック内部で宣言された変数の有効範囲はブロックの最後までです。

```
{ 文 または 変数宣言 ... }
```

3.10.4 if 文

条件によって実行の流れを変更するための文です。「条件式」は論理型の値を返す式でなければなりません。「条件式」の値が true の場合「本文 1」が実行され、false の場合「本文 2」が実行されます。なお、else 節 (「else 本文 2」の部分) は省略することができます。

```
if ( 条件式 ) 本文 1 【 else 本文 2 】
```

3.10.5 while 文

条件が成立している間、while 文の「本文」を実行します。「条件式」は論理型の値を返す式でなければなりません。まず、「条件式」を計算し、値が true なら「本文」が実行されます。これは、「条件式」の値が false になるまで繰り返されます。

```
while ( 条件式 ) 本文
```

3.10.6 do-while 文

条件が成立している間、do-while 文の「本文」を実行します。「条件式」は論理型の値を返す式でなければなりません。まず「本文」を実行し、次に「条件式」を計算します。「条件式」の値が true なら、再度、「本文」の実行に戻ります。これは、「条件式」の値が false になるまで繰り返されます。

```
do 本文 while ( 条件式 ) ;
```

3.10.7 for 文

便利に拡張された while 文です。「条件式」は論理型でなければなりません。まず、「初期化式」か「ローカル変数宣言」を実行します。次に「条件式」を計算し、値が true なら「本文」を実行します。最後に「再初期化式」を実行し、その後「条件式」の計算に戻ります。これは、「条件式」の値が false になるまで繰り返されます。

「ローカル変数宣言」で宣言された変数は、「条件式」、「再初期化式」、「本文」で使用することができますが、それ以降では使用できません。「初期化式」、「条件式」、「再初期化式」のどれも省略可能です。「条件式」を省略した場合は無限ループの記述になります。

for(【初期化式｜ローカル変数宣言】;【条件式】;【再初期化式】) 本文

```
// for 文の使用例
int i;
for (i=0; i<10; i=i+1) { ... }
n = i;                                  // i をここで使用できる

for (int j=0; j<10; j=j+1) { ... }
n = j;                                  // （エラー）j が未定義になる

for (;;) { ... }                        // 無限に本文を繰り返す
```

3.10.8 return 文

関数から戻るときに使用します。関数の途中で使用すると、関数の途中から呼び出し側に戻ることができます。void 型以外の関数では、関数の最後の文が return 文でなければなりません。「式」は void 型の関数では書いてはなりません。逆に、void 型以外の関数では書かなければなりません。「式」の型と関数の型は一致していなければなりません。なお、interrupt 関数には void 型の関数と同じルールが適用されます。

return 【 式 】 ;

```
// void 型以外の関数
int f() {
  ...
  if (err) return 1;      // f の途中から戻る
  ...
  return 0;               // この return は省略できない
}
// void 型の関数
void g() {
  ...
  if (err) return;        // g の途中から戻る
  ...
  return;                 // この return は省略しても良い
}
```

3.10.9 break 文

for 文や while 文、do-while 文の繰り返しから脱出します。多重ループから一度に脱出することはできません。

```
break;
```

3.10.10 continue 文

for 文や while 文、do-while 文の本文の実行をスキップします。for 文では再初期化式に、while 文と do-while 文では条件式にジャンプします。

```
continue;
```

第4章　ライブラリ関数

C--言語で使用できる関数です。必要最低限の関数が、TaC 版、C 言語トランスレータ版で使用できます。

4.1　標準入出力ライブラリ

#include <stdio.hmm>を書いた後で使用します。トランスレータ版では C 言語の高水準 I/O 関数の呼出しに変換されます。TaC 版でも入出力の自動的なバッファリングを行います。TaC 版ではバッファサイズは 128 バイトです。以下の関数が使用できます。

4.1.1　printf 関数

標準出力ストリームに format 文字列を用いた変換付きで出力します。出力した文字数を関数の値として返します。

```
#include <stdio.hmm>
public int printf(char[] format, ...);
```

トランスレータ版では C 言語の printf 関数の呼出しに変換されます。TaC 版では簡易 printf 関数を使用できます。TaC 版の簡易 printf 関数では、format 文字列に以下のような変換を記述できます。

%[-][数値] 変換文字

-を書くと左詰めで表示します。数値は表示に使用するカラム数を表します。数値を 0 で開始した場合は、数値の右づめ表示で空白の代わりに 0 が用いられます。使用できる変換文字は次の表の通りです。

変換文字	意味
o	整数値を 8 進数で表示する
d	整数値を 10 進数で表示する
x	整数値を 16 進数で表示する
c	ASCII コードに対応する文字を表示する
s	文字列を表示する
%	%を表示する

4.1.2　puts 関数

標準出力ストリームへ 1 行出力します。エラーが発生した場合は null を、正常時には s を返します。

```
#include <stdio.hmm>
public char[] puts(char[] s);
```

4.1.3　putchar 関数

標準出力ストリームへ 1 文字出力します。エラーが発生した場合は true を、正常時には false を返します。

```
#include <stdio.hmm>
public boolean putchar(char c);
```

4.1.4　getchar 関数

標準入力ストリームから 1 文字入力します。C 言語の getchar 関数と異なり char 型なので EOF チェックができません。現在のところ、TacOS では標準入力を EOF にする方法は準備されていません。

```
#include <stdio.hmm>
public char getchar();
```

4.1.5　fopen 関数

ファイルを開きます。path 引数はファイルへのパス、mode 引数はオープンのモードです。パスは “/” 区切りで表現します。TacOS にはカレントディレクトリはありません。fopen は正常に FILE 構造体、エラー時に null を返します。

```
#include <stdio.hmm>
public FILE fopen(char[] path, char[] mode);
```

mode の意味は次の通りです。

mode	意味
"r"	読み込みモードで開く
"w"	書き込みモードで開く（ファイルが無ければ作る）
"a"	追記モードで開く（ファイルが無ければ作る）

fopen は、書き込みモード、追記モードの時ファイルが存在しない場合、自動的にファイルを作成します。

4.1.6 fclose 関数

ストリームをクローズします。TacOS では、標準入出力ストリーム（stdin、stdout、stderr）をクローズすることはできません。fclose は正常に false、エラー時に true を返します。

```
#include <stdio.hmm>
public boolean fclose(FILE stream);
```

4.1.7 fprintf 関数

出力ストリームを明示できる printf 関数です。stream 引数に出力先を指定します。出力ストリームは、fopen で開いたファイルか stdout です。

```
#include <stdio.hmm>
public int fprintf(FILE stream, char[] format, ...);
```

4.1.8 fputs 関数

出力ストリームを明示できる puts 関数です。stream 引数に出力先を指定します。出力ストリームは、fopen で開いたファイルか stdout です。

```
#include <stdio.hmm>
public char[] fputs(char[] s, FILE stream);
```

4.1.9 fputc 関数

出力ストリームを明示できる putchar 関数です。stream 引数に出力先を指定します。出力ストリームは、fopen で開いたファイルか stdout です。

```
#include <stdio.hmm>
public boolean fputc(char c, FILE stream);
```

4.1.10 fgets 関数

任意の入力ストリームから 1 行入力します。入力は buf に文字列として格納します。n 引数には buf のサイズを渡します。通常、buf には\n も格納されます。fgets は、EOF で null を、正常時には buf を返します。

```
#include <stdio.hmm>
public char[] fgets(char[] buf, int n, FILE stream);
```

C--では、C 言語の gets が使用できません。gets はバッファオーバーフローの危険があるので C--には持込みませんでした。C--で、gets を使用したい時は fgets を使用して次のように書きます。

```
while (fgets(buf, N, stdin)!=null) { ...
```

4.1.11　fgetc 関数

任意の入力ストリームから 1 文字入力します。C 言語の fgetc 関数と異なり char 型なので EOF チェックができません。TaC 版では安全のため、fgetc 関数が EOF に出会うと強制的にプログラムを終了する仕様になっています。fgetc 関数を実行する前に、必ず、feof 関数を用いて EOF チェックをする必要があります。

```
#include <stdio.hmm>
public char fgetc(FILE stream);
```

4.1.12　feof 関数

入力ストリームが EOF になっていると true を返します。fgetc を実行する前に EOF チェックのために使用します。

```
#include <stdio.hmm>
public boolean feof(FILE stream);
```

4.1.13　ferror 関数

ストリームがエラーを起こしていると true を返します。

```
#include <stdio.hmm>
public boolean ferror(FILE stream);
```

4.1.14　fflush 関数

出力ストリームのバッファをフラッシュします。入力ストリームをフラッシュすることはできません。正常時 false、エラー時 true を返します。stderr はバッファリングされていないので、フラッシュしても何も起きません。

```
#include <stdio.hmm>
public boolean fflush(FILE stream);
```

4.1.15 readDir 関数

ディレクトリファイルを読みます。TacOS 版だけで使用できる関数です。fd には open システムコールでオープン済のファイル記述子を、dir には Dir 構造体のインスタンスを渡します。Dir 構造体の name メンバーは、大きさ 12 の文字配列で初期化されている必要があります。

ls プログラムのソースコードに使用例があります。

```
#include <stdio.hmm>
struct Dir {
    char[] name;
    int    attr;
    int    clst;
    int    lenH, lenL;
};
Dir dir = {"            ", 0, 0, 0, 0 };
public int readDir(int fd, Dir dir);
```

4.1.16 perror 関数

現在の errno グローバル変数の値に応じたエラーメッセージを表示します。msg はエラーメッセージの先頭に付け加えます。errno にはシステムコールやライブラリ関数がエラー番号をセットします。表 4.1 に TaC 版のエラーとメッセージの一覧を示します。

```
#include <stdio.hmm>
#include <errno.hmm>
public void perror(char[] msg);
```

4.1.17 プログラム例

C--言語で記述した、標準入出力関数の使用例を以下に示します。

TacOS 専用版のプログラム例

TacOS では、errno 変数にセットされるエラー番号が負の値になっています。また、アプリケーションが負の終了コードで終わった場合、シェルがエラーメッセージを表示する仕様になっています。

更にライブラリは、ユーザプログラムのバグが原因と考えられるエラーや、メモリ不足のような対処が難しいエラーが発生したとき、負の終了コードでプログラムを終了します。そこで、次のようなエラー処理を簡略化したプログラムを書くことができます。

```
// ファイルの内容を表示するプログラム
//  (TacOS 専用バージョン)
#include <stdio.hmm>
#include <errno.hmm>
public int main(int argc, char[][] argv) {
  FILE fp = fopen("a.txt", "r");
  if (fp==null) exit(errno);     // エラー表示をシェルに任せる
  while (!feof(fp))
    putchar(fgetc(fp));
  fclose(fp);
  return 0;
}
```

TacOS トランスレータ共通版のプログラム例

C 言語プログラム風に記述することもできます。この例では、fopen がエラーになったファイルの名前をエラーメッセージに含めることができます。

```
// ファイルの内容を表示するプログラム
//  (トランスレータ、TacOS 共通バージョン)
#include <stdio.hmm>
public int main(int argc, char[][] argv) {
  char fname = "a.txt";
  FILE fp = fopen(fname, "r");
  if (fp==null) {
    perror(fname);     // エラー表示を自分で行う
    return 1;
  }
  while (!feof(fp))
    putchar(fgetc(fp));
  fclose(fp);
  return 0;
}
```

表 4.1: エラー一覧

記号名	メッセージ	意味
ENAME	Invalid file name	ファイル名が不正
ENOENT	No such file or directrory	ファイルが存在しない
EEXIST	File exists	同名ファイルが存在する
EOPEND	File is opened	既にオープンされている
ENFILE	File table overflow	システム全体のオープン数超過
EBADF	Bad file number	ファイル記述子が不正
ENOSPC	No space left on device	デバイスに空き領域が不足
EPATH	Bad path	パスが不正
EMODE	Bad mode	モードが一致しない
EFATTR	Bad attribute	ファイルの属性が不正
ENOTEMP	Directory is not empty	ディレクトリが空でない
EINVAL	Invalid argument	引数が不正
EMPROC	Process table overflow	プロセスが多すぎる
ENOEXEC	Bad EXE file	EXE ファイルが不正
EMAGIC	Bad MAGIC number	不正なマジック番号
EMFILE	Too many open files	プロセス毎のオープン数超過
ECHILD	No children	子プロセスが存在しない
ENOZOMBI	No zombie children	ゾンビ状態の子が存在しない
ENOMEM	Not enough memory	十分な空き領域が無い
ESYSNUM	Invalid system call number	システムコール番号が不正
EZERODIV	Zero division	ゼロ割り算
EPRIVVIO	Privilege violation	特権違反
EILLINST	Illegal instruction	不正命令
EUSTK	Stack overflow	スタックオーバーフロー
EUMODE	stdio: Bad open mode	モードと使用方法が矛盾
EUBADF	stdio: Bad file pointer	不正な fp が使用された
EUEOF	fgetc: EOF was ignored	fgetc 前に EOF チェック必要
EUNFILE	fopen: Too many open files	プロセス毎のオープン超過
EUSTDIO	fclose: Standard i/o should not be closed	標準 io はクローズできない
EUFMT	fprintf: Invalid conversion	書式文字列に不正な変換
EUNOMEM	malloc: Insufficient memory	ヒープ領域が不足
EUBADA	free: Bad address	malloc した領域ではない

4.2　標準ライブラリ

#include <stdlib.hmm>を書いた後で使用します。

4.2.1　malloc 関数

ヒープ領域に size バイトのメモリ領域を確保し、領域を指す参照を返します。malloc 関数は void[] 型なので、領域を指す参照は全ての参照変数に代入できます。

TaC 版では、ヒープ領域に十分な空きが見つからないとき終了コード EUNOMEM でプログラムを終了します。トランスレータ版では、エラーメッセージを表示したあと終了コード 1 でプログラムを終了します。

```
#include <stdlib.hmm>
public void[] malloc(int size);
```

4.2.2　free 関数

malloc 関数で割当てた領域を解放します。TaC 版では、領域が malloc 関数で割当てたものではない可能性がある場合（マジックナンバーが破壊されている、管理されている空き領域と重なる等)、終了コード EUBADA でプログラムを終了します。

```
#include <stdlib.hmm>
public void free(void[] mem);
```

4.2.3　atoi 関数

atoi 関数は引数に渡した 10 進数文字列を解析して、それが表現する値を返します。

```
#include <stdlib.hmm>
public int atoi(char[] s);
```

4.2.4　srand 関数

srand 関数は擬似乱数発生器を seed で初期化します。

```
#include <stdlib.hmm>
public void srand(int seed);
```

4.2.5 rand 関数

rand 関数は次の擬似乱数を発生します。

```
#include <stdlib.hmm>
public int rand();
```

4.2.6 exit 関数

exit 関数はオープン済みのストリームをフラッシュしてからプログラムを終了します。status は、親プロセスに返す終了コードです。0 が正常終了の意味、1 以上はユーザが決めた終了コードです。

TacOS 版では、負の終了コードが使用できます。使用できるコードは表 4.1 に記号定数として定義されています。負の値を返すと親プロセスがシェルの場合、シェル側でエラーメッセージを表示してくれます。

```
#include <stdlib.hmm>
public void exit(int status);
```

4.3 文字列操作関数

#include <string.hmm>を書いた後で使用します。

4.3.1 strCpy 関数

文字列 s を文字配列 d にコピーし、d を関数値として返します。トランスレータ版では C 言語の strcpy 関数に置き換えられます。

```
#include <string.hmm>
public char[] strCpy(char[] d, char[] s);
```

4.3.2 strNcpy 関数

文字列 s の最大 n 文字を文字配列 d にコピーし、d を関数値として返します。文字配列の使用されない部分には'\0' が書き込まれます。文字列の長さが n 以上の場合は、'\0' が書き込まれないので注意して下さい。トランスレータ版では C 言語の strncpy 関数に置き換えられます。

```
#include <string.hmm>
public char[] strNcpy(char[] d, char[] s, int n);
```

4.3.3　strCat 関数

文字列 s を文字配列 d に格納されている文字列の後ろに追加し、d を関数値として返します。トランスレータ版では C 言語の strcat 関数に置き換えられます。

```
#include <string.hmm>
public char[] strCat(char[] d, char[] s);
```

4.3.4　strNcat 関数

文字列 s の先頭 n 文字未満を、文字配列 d に格納されている文字列の後ろに追加し、d を関数値として返します。d に格納された文字列の最後には'\0' が書き込まれます。トランスレータ版では C 言語の stnrcat 関数に置き換えられます。

```
#include <string.hmm>
public char[] strNcat(char[] d, char[] s, int n);
```

4.3.5　strCmp 関数

文字列 s1 と文字列 s2 を比較します。strCmp 関数は、アスキーコード順で s1 が大きいとき正の値、s1 が小さいとき負の値、同じ時 0 を返します。トランスレータ版では C 言語の strcmp 関数に置き換えられます。

```
#include <string.hmm>
public int strCmp(char[] s1, char[] s2);
```

4.3.6　strNcmp 関数

文字列 s1 と文字列 s2 の先頭 n 文字を比較します。strNcmp 関数は、strcmp 関数同様にアスキーコード順で大小を判断します。トランスレータ版では C 言語の strncmp 関数に置き換えられます。

```
#include <string.hmm>
public int strNcmp(char[] d, char[] s, int n);
```

4.3.7　strLen 関数

文字列 s の長さを返します。長さに'\0' は含まれません。トランスレータ版では C 言語の strlen 関数に置き換えられます。

```
#include <string.hmm>
public int strLen(char[] s);
```

4.3.8 strChr 関数

文字列 s の中で最初に文字 c が現れる位置を、s 文字配列の添字で返します。文字 c が含まれていない場合は-1 を返します。トランスレータ版では lib/wrapper.c の関数に置き換えられます。

```
#include <string.hmm>
public int strChr(char[] s, char c);
```

4.3.9 strRchr 関数

文字列 s の中で最後に文字 c が現れる位置を、s 文字配列の添字で返します。文字 c が含まれていない場合は-1 を返します。トランスレータ版では lib/wrapper.c の関数に置き換えられます。

```
#include <string.hmm>
public int strRchr(char[] s, char c);
```

4.3.10 strStr 関数

文字列 s1 の中に文字列 s2 が現れる位置を、s1 文字配列の添字で返します。文字列 s2 が含まれていない場合は-1 を返します。トランスレータ版では lib/wrapper.c の関数に置き換えられます。

```
#include <string.hmm>
public int strStr(char[] s1, char[] s2);
```

4.3.11 subStr 関数

文字列 s の先頭 pos 文字を省いた部分文字列を返します。返した文字列は s とメモリ領域を共用していますので注意が必要です。トランスレータ版では lib/wrapper.c の関数に置き換えられます。

```
#include <string.hmm>
public char[] subStr(char[] s, int pos);
```

4.4 文字クラス分類関数

#include <ctype.hmm>を書いた後で使用します。

4.4.1 isAlpha 関数

文字 c がアルファベットなら true を返します。

```
#include <ctype.hmm>
public boolean isAlpha(char c);
```

4.4.2 isDigit 関数

文字 c が数字なら true を返します。

```
#include <ctype.hmm>
public boolean isDigit(char c);
```

4.4.3 isAlnum 関数

文字 c がアルファベットか数字なら true を返します。

```
#include <ctype.hmm>
public boolean isAlnum(char c);
```

4.4.4 isPrint 関数

文字 c が印刷可能文字なら true を返します。

```
#include <ctype.hmm>
public boolean isPrint(char c);
```

4.4.5 isLower 関数

文字 c がアルファベット小文字なら true を返します。

```
#include <ctype.hmm>
public boolean isLower(char c);
```

4.4.6 isUpper 関数

文字 c がアルファベット大文字なら true を返します。

```
#include <ctype.hmm>
public boolean isUpper(char c);
```

4.4.7 isXdigit 関数

文字 c が 16 進数文字なら true を返します。

```
#include <ctype.hmm>
public boolean isXdigit(char c);
```

4.4.8 isSpace 関数

文字 c が空白文字（'\t'、'\n'、'\v'、'\f'、'\r'、' '）なら true を返します。

```
#include <ctype.hmm>
public boolean isSpace(char c);
```

4.4.9 toLower 関数

文字 c がアルファベット大文字なら小文字に変換して返します。文字 c がアルファベット大文字以外の場合は変換しないで返します。

```
#include <ctype.hmm>
public char toLower(char c);
```

4.4.10 toUpper 関数

文字 c がアルファベット小文字なら大文字に変換して返します。文字 c がアルファベット小文字以外の場合は変換しないで返します。

```
#include <ctype.hmm>
public char toUpper(char c);
```

4.5 特殊な関数

C--言語にはキャスティング演算や、ポインタ演算がありません。これらの代用となる関数が#include <crt0.hmm>を書いた後で使用できます。

4.5.1 _iToA 関数

整数から参照へ型を変換する関数です。整数を引数に void[] 参照 (アドレス) を返します。トランスレータ版で 64bit の C 言語環境では、int 型とポインタ型のビット数が異な

るので使用できません。関数の値は void[] 型の参照なので、どのような参照型変数にも代入できます。

```
#include <crt0.hmm>
public void[] _iToA(int a);
```

4.5.2　_aToI 関数

参照から整数へ型を変換する関数です。参照 (アドレス) を引数に整数を返します。トランスレータ版で 64bit の C 言語環境では、int 型とポインタ型のビット数が異なるので使用できません。引数の型は void[] なので、参照型ならどんな型でも渡すことができます。

```
#include <crt0.hmm>
public int _aToI(void[] a);
```

4.5.3　_aToA 関数

参照から参照へ型を変換する関数です。異なる型の参照の間で代入をするために使用できます。

```
#include <crt0.hmm>
public void[] _aToA(void[] a);
```

4.5.4　_addrAdd 関数

C 言語のポインタ演算の代用にする関数です。参照 (アドレス) と整数を引数に渡し、参照から整数バイト先の参照 (アドレス) を返します。

```
#include <crt0.hmm>
public void[] _addrAdd(void[] a, int n);
```

4.5.5　_aCmp 関数

参照 (アドレス) の大小比較を行う関数です。C--言語では参照の大小比較はできません。Java 言語でも参照の大小比較はできないので、通常はこの仕様で十分と考えられます。しかし、malloc、free 関数等の実現にはアドレスの大小比較が必要です。そこで、アドレスの大小比較をする _aCmp 関数を用意しました。_aCmp 関数は、a の方が大きい場合は 1 を、b の方が大きい場合は -1 を、a と b が等しい場合は 0 を返します。

```
#include <crt0.hmm>
public int _aCmp(void[] a, void[] b);
```

4.5.6 _uCmp 関数

符号無し数の比較を行う関数です。_uCmp 関数は、a の方が大きい場合は 1 を、b の方が大きい場合は -1 を、a と b が等しい場合は 0 を返します。

```
#include <crt0.hmm>
public int _uCmp(int a, int b);
```

4.5.7 _args 関数

printf 関数のような可変個引数の関数を実現するために、可変個引数関数の内部で引数を配列としてアクセスできるようにする関数です。_args 関数は_args を呼び出した C--関数の第 1 引数を添字 0 とする int 配列を返します。_args 関数は、C 言語トランスレータ版では使用できません。

```
#include <crt0.hmm>
public int[] _args();
```

次に可変個引数関数の記述例を示します。

```
int f(char[] s, ...) {          // ... は可変個引数の関数を表す
  int[] args = _args();         // args 配列は引数配列を格納
  printf("%s\n", args[0]);      // 引数 s のこと（第 1 引数）
  printf("%d\n", args[1]);      // 引数 ... の最初に該当（第 2 引数）
  printf("%d\n", args[2]);      // 引数 ... の 2 番に該当（第 3 引数）
```

第5章　システムコール

TacOS のシステムコールを呼び出す関数です。**トランスレータ版では使用できません。**
`#include <syslib.hmm>`と書いた後で使用します。

5.1　プロセス関連

TacOS では、exec で新しいプロセスを作ると同時に新しいプログラムを実行します。UNIX の fork-exec 方式とは異なります。

5.1.1　exec

name でファイル名を指定して新しいプロセスでプログラムの実行を開始します。argv は、開始するプログラムの main 関数の第２引数（char[][]argv）に渡される文字列配列です。exec は正常なら 0、エラー発生なら負のエラー番号を返します。

```
#include <syslib.hmm>
public int exec(char[] name, void[] argv);
```

argv[0] にプログラム名、argv[1] に第 1 引数のように格納します。最後に null を格納します。次に使用例を示します。

```
char[][] args = {"prog", "param1", "param2", null};
void f() {
  exec("/bin/prog.exe", args);
}
```

子プロセス側は次のようなプログラムになります。

```
int main(int argc, char[][]argv) {
  int c = argc;        // 前のプログラムで起動されたとき 3
  char[] s = argv[1]; // 前のプログラムで起動されたとき "param1"
  return 0;
}
```

5.1.2 _exit

_exit はプログラム（プロセス）を終了します。_exit は入出力のバッファをフラッシュしません。_exit は緊急終了用に使用し、普通は exit を使用します。

status は、親プロセスに返す終了コードです。0 が正常終了の意味、1 以上はユーザが決めた終了コード、負の値は表 4.1 に示す記号名で定義されています。負の値を返すと親プロセスがシェルの場合、シェル側でエラーメッセージを表示してくれます。

```
#include <syslib.hmm>
public void _exit(int status);
```

5.1.3 wait

wait は子プロセスの終了を待ちます。stat には大きさ 1 の int 配列を渡します。子プロセスが終了した際、stat[0] に終了コードが書き込まれます。wait は正常なら 0、エラー発生なら負のエラー番号を返します。

```
#include <syslib.hmm>
public int wait(int[] stat);
```

5.1.4 sleep

sleep はプロセスを指定された時間だけ停止します。ms はミリ秒単位で停止時間を指定します。ms に負の値を指定すると EINVAL エラーになります。それ以外では、sleep は 0 を返します。

```
#include <syslib.hmm>
public int sleep(int ms);
```

5.2 ファイル操作

TacOS は、microSD カードの FAT16 ファイルシステムを扱うことができます。VFAT には対応していません。

5.2.1 creat

creat は新規ファイルを作成します。path は新しいファイルのパスです。creat は正常なら 0、エラー発生なら負のエラー番号を返します。

```
#include <syslib.hmm>
public int creat(char[] path);
```

5.2.2 remove

remove はファイルを削除します。path は削除するファイルのパスです。remove は正常なら 0、エラー発生なら負のエラー番号を返します。

```
#include <syslib.hmm>
public int remove(char[] path);
```

5.2.3 mkDir

mkDir は新規のディレクトリを作成します。path は新しいディレクトリのパスです。mkDir は正常なら 0、エラー発生なら負のエラー番号を返します。

```
#include <syslib.hmm>
public int mkDir(char[] path);
```

5.2.4 rmDir

rmDir はディレクトリを削除します。path は削除するディレクトリのパスです。rmDir は正常なら 0、エラー発生なら負のエラー番号を返します。削除するディレクトリが空でない場合はエラーになります。

```
#include <syslib.hmm>
public int rmDir(char[] path);
```

5.3 ファイルの読み書き

ファイルの読み書きには、通常は 33 ページの標準入出力ライブラリ関数を用います。以下のシステムコールは、主にライブラリ関数の内部で使用されます。

5.3.1 open

open はファイルを開きます path は開くファイルのパスです。mode には READ、WRITE、APPEND のいずれかを指定します。open は正常なら非負のファイル記述子、エラー発生なら負のエラー番号を返します。ファイルが存在しない場合は、どのモードでもエラーになります。新規ファイルに書き込みたい場合は、事前に creat システムコールを用いてファイルを作成しておく必要があります。

open はディレクトリを READ モードで開くことができます。ディレクトリは 37 ページの readDir 関数で読みます。

```
#include <syslib.hmm>
public int open(char[] path, int mode);
```

5.3.2 close

close は open で開いたファイルを閉じます。fd は閉じるファイルのファイル記述子です。close は正常なら 0、エラー発生なら負のエラー番号を返します。

```
#include <syslib.hmm>
public int close(int fd);
```

5.3.3 read

read は open を用い READ モードで開いたファイルからデータを読みます。fd はファイル記述子です。buf はデータを読み込むバッファ、len はバッファサイズ（バイト単位）です。read は正常なら読み込んだバイト数、エラー発生なら負のエラー番号を返します。EOF では 0 を返します。

```
#include <syslib.hmm>
public int read(int fd, void[] buf, int len);
```

5.3.4 write

write は open を用い WRITE、APPEND モードで開いたファイルからデータを読みます。fd はファイル記述子です。buf は書き込むデータが置いてあるバッファ、len は書き込むデータのサイズ（バイト単位）です。write は正常なら書き込んだバイト数、エラー発生なら負のエラー番号を返します。

```
#include <syslib.hmm>
public int write(int fd, void[] buf, int len);
```

5.3.5 seek

seek は open を用い開いたファイルの読み書き位置を変更します。fd はファイル記述子です。seek 位置は、上位 16bit（ptrh）と下位 16bit（ptrl）を組み合わせて指定します。seek は正常なら 0、エラー発生なら負のエラー番号を返します。

```
#include <syslib.hmm>
public int seek(int fd, int ptrh, int ptrl);
```

5.4 コンソール関連

コンソール入出力には、通常は 33 ページの標準入出力ライブラリ関数を用います。以下のシステムコールは、主にライブラリ関数の内部で使用されます。conRead はライブラリ関数が stdin からの読み込みをする場合に、conWrite はライブラリ関数が stdout、stderr への書き込みをする場合にライブラリ関数内部で使用されます。

5.4.1 conRead

conRead はキーボードから 1 行入力します。読み込んだ内容は buf で指定されるバッファに文字列として格納されます。len は buf のバイト数です。文字列の最後には'\0' が格納されます。'\n' は含まれません。buf の大きさは入力可能な文字数より 1 大きくする必要があります。conRead はキーボードから入力した文字数を返します。

```
#include <syslib.hmm>
public int conRead(char[] buf, int len);
```

5.4.2 conWrite

conWrite は画面に文字列を出力します。buf には出力する文字列を格納して渡します。

```
#include <syslib.hmm>
public int conWrite(char[] buf);
```

conWrite は以下の制御文字を解釈します。

制御文字	働き
'\r'	カーソルを現在行の左端に移動する
'\n'	カーソルを次の行の左端に移動する
'\t'	カーソルを次の TAB ストップに移動する
'\x08'	カーソルを右に 1 文字分移動する
'\x0c'	画面をクリアしカーソルを画面右上端に移動する

付 録A　C--言語文法まとめ

以下に、C--言語の文法を BNF 風にまとめたものを掲載します。仕様の最後に掲載されている “#ディレクティブ” は、プリプロセッサがソースプログラムのファイル名を C--コンパイラに伝えるために出力するものです。C--コンパイラは、“#ディレクティブ” によって、ソースプログラムやインクルードファイルの名前を知ることができます。

次ページで用いる文法表記方法
　全角記号がメタ文字、意味は次の通り
　(1) A ※　 は A のゼロ回以上の繰り返し
　(2) 《...》はグループ
　(3) 【...】は省略可能
　(4) A ｜ B は A または B

```
プログラム：      《 構造体宣言 | だまし型宣言 | 大域変数定義 | 関数定義 |
                  関数宣言 》※
構造体宣言：      struct 名前 { 《 型 名前 《 , 名前 》※ ; 》※ } ;
だまし型宣言：    typedef 名前 ;
大域変数定義：    【修飾子】 型 大域変数並び ;
大域変数並び：    名前 【 = 初期化 】《 , 名前 【 = 初期化 】 》※
修飾子：          public
型：              int | char | boolean |void | interrupt | 名前 | 型 []
初期化：          定数 | { 初期化 《 , 初期化 》※ } |
                  array( 数値 《 , 数値 》※ )
定数：            数値 | 文字列 | null | true |false
関数定義：        【修飾子】 型 名前 ( 引数リスト ) ブロック
関数宣言：        【修飾子】 型 名前 ( 引数リスト ) ;
引数リスト：      【 型 名前 《 , 型 名前 》※ 【 , ... 】 】 | ...
ブロック：        { 《 局所変数定義 | 文 》※ }
局所変数定義：    型 局所変数並び ;
局所変数並び：    名前 【 = 代入式 】《 , 名前 【 = 代入式 】 》※

文：              IF 文 | WHILE 文 | DO-WHILE 文 | FOR 文 | RETURN 文 |
                  BREAK 文 | CONTINUE 文 | ブロック | 式 ; | ;
IF 文：           if ( 式 ) 文 【 else 文 】
WHILE 文：        while ( 式 ) 文
DO-WHILE 文:      do 文 while ( 式 ) ;
FOR 文：          for( 【 式 | 局所変数定義 】 ; 【 式 】 ; 【 式 】 ) 文
RETURN 文：       return 【 式 】;
BREAK 文：        break;
CONTINUE 文：     continue;

式：              代入式 | 式 , 代入式
代入式：          論理 OR 式 | 論理 OR 式 = 代入式
論理 OR 式：      論理 AND 式 | 論理 OR 式 || 論理 AND 式
論理 AND 式：     OR 式 | 論理 AND 式 && OR 式
OR 式：           XOR 式 | OR 式 | XOR 式
XOR 式：          AND 式 | XOR 式 ^ AND 式
AND 式：          等式 | AND 式 & 等式
等式：            比較式 | 等式 == 比較式 | 等式 != 比較式
比較式：          シフト式 | 比較式 < シフト式 |比較式 <= シフト式 |
                  比較式 > シフト式 | 比較式 >= シフト式
シフト式：        和式 | シフト式 << 和式 | シフト式 >> 和式
和式：            積式 | 和式 + 積式 | 和式 - 積式
積式：            単項式 | 積式 * 単項式 |積式 / 単項式 | 積式 % 単項式
単項式：          因子 | + 単項式 | - 単項式 |! 単項式 | ~ 単項式
因子：            名前 | 定数 | 関数呼出 | ( 式 ) |因子 [ 代入式 ] |
                  因子. 名前 | sizeof( 型 ) |addrof( 名前 ) | ord( 式 ) |
                  chr( 式 ) | bool( 式 )
関数呼出：        名前 ( 【 代入式 《 , 代入式 》※ 】 )

その他
コメント：        /* ... */ または // ...
ディレクティブ：  # 行番号 "ファイル名"
```

付 録B　コマンドリファレンス

UNIX や MacOS 上で動作する C--コンパイラの使用方法を解説します。

B.1　c--コマンド

c--コマンドは C--言語の TaC 用コンパイラです。

c--コマンドは、C--言語で記述されたプログラムを入力し、TaC アセンブリ言語で記述したプログラムに変換します。c--コマンドの書式は次の通りです。

形式：　c-- [-h] [-v] [-O0] [-O] [-O1] [-K] [<source file>]

引数に C--言語のソースプログラムファイルを指定した場合は、指定されたファイルからソースプログラムを読み込みます。ファイルが省略された場合は標準入力ストリームからソースプログラムを読み込みます。どちらの場合もコンパイル結果は標準出力ストリームに出力します。ソースプログラムファイルの拡張子は「.cmm」にします。

-h、-v オプションは使用法メッセージを表示します。-O0 オプションを指定すると、ソースコード中の定数式をコンパイル時に計算したり、実行されることがないプログラムの部分を削除したりする等の最適化をしません。-O、-O1 は最適化を促すオプションですが、デフォルトで ON になっているので指定する必要はありません。-K オプションを使うと、関数入口へのスタックオーバーフロー検出コードの埋め込みが抑制されます。TacOS のカーネルをコンパイルするときに使用するオプションです。

c--コマンドは、C 言語のプリプロセッサと組み合わせて次のように使用します。コンパイル結果はリダイレクトを用いて拡張子「.s」のファイルに出力します。

```
$ cc -E -std=c99 -nostdinc -I/usr/local/cmmInclude \
-I/usr/local/cmmLib - < hello.cmm | c-- > hello.s
```

この実行例は、OSX 10.11 の場合です。OSX 10.11 の cpp コマンドは、//コメントをうまく処理できないようです。cc -E コマンドを代用にするとうまく処理できます。他の実行環境では、その環境にあう実行方法を試行錯誤する必要があるかもしれません。

B.2 vm-c--コマンド

仮想スタックマシンのニーモニックを出力するC--コンパイラです。

仮想スタックマシンのニーモニックは、コンパイラ内部で用いている中間言語（61 ページ参照）と、ほぼ一対一に対応します。中間言語や仮想スタックマシンを学習したいときにこのコマンドを利用します。vm-c--コマンドの書式は次の通りです。

形式： `vm-c-- [-h] [-v] [-O0] [-O] [-O1] [-K] [<source file>]`

引数の意味はc--コマンドと同様です。次の実行例は変換結果を画面に表示しています。

```
$ cc -E -std=c99 -nostdinc -I/usr/local/cmmInclude \
 - < hello.cmm | vm-c--
_stdin
        WS      1
_stdout
        WS      1
_stderr
        WS      1
.L1
        STRING  "hello,world\n"
_main
        ENTRY   0
        LDC     .L1
        ARG
        CALLF   1,_printf
        POP
        LDC     0
        MREG
        RET
```

B.3 c-c--コマンド

C言語に変換して出力するトランスレータです。

出力されたC言語は、/user/local/cmmLib/ にインストールされた、cfunc.hmm や wrapper.c を用いて、UNIX や MacOS で実行できるようになります。

引数の意味はc--コマンドと同様です。次の実行例は変換結果を画面に表示しています。

```
$ cc -E -std=c99 -nostdinc -I/usr/local/cmmInclude \
 -I/usr/local/cmmLib - < hello.cmm | c-c--
#include <stdio.h>
#define _cmm_str_L0 "hello,world\n"
int main(){
printf(_cmm_str_L0);
}
```

次の実行例は変換結果を hello.c ファイルに格納したあと、C 言語コンパイラでコンパイルして実行した例です。

```
$ cc -E -std=c99 -nostdinc -I/usr/local/cmmInclude \
 -I/usr/local/cmmLib - < hello.cmm | c-c-- > hello.c
$ cc -o hello -Wno-parentheses-equality hello.c \
 /usr/local/cmmLib/wrapper.c
$ ./hello
hello,world
```

付 録C　中間言語

C--コンパイラに入力されたC--プログラムは、一旦、以下で説明する中間言語に変換されます。その後、中間言語から仮想のスタックマシンやTaCのニーモニックに変換されます。なお、C言語トランスレータは構文木から直接C言語を生成するので、中間言語を用いません。

C.1　仮想スタックマシン

以下では中間言語を変換する先として仮想のスタックマシンを想定します。仮想スタックマシンの命令は、中間言語から、ほぼ、一対一に変換できます。仮想スタックマシンは、次のようなものです。

- 仮想スタックマシンが扱うデータは、基本的にワードデータ (16bit) です。C--言語のint型、参照（アドレス）はワードデータにピッタリ格納されます。char型はワードデータの下位8bit、boolean型はワードデータの下位の1bitに格納されます。
- char型とboolean型の**配列要素だけ**、メモリ節約のためバイトデータ (8bit) です。LDB命令、STB命令がバイト配列のデータをアクセスします。
- 仮想スタックマシンのプログラムは次の書式のニーモニックで記述します。

```
[ラベル]　　[命令　[オペランド[,オペランド]...]]
```

C.2　書式

中間言語は次のような命令行で表現されます。

```
命令([オペランド[,オペランド]...])

例： vmLdCns(3)　　// 定数3をスタックに積む
```

C.3　命令

中間言語の命令には「ラベル生成命令」、「マシン命令」、「マクロ命令」、「擬似命令」があります。

C.3.1 ラベル生成命令

プログラムのジャンプ先やデータのためにラベルを生成します。

vmName

名前を表現するラベルを宣言します。名前表の idx 番目に登録されている名前をラベルとして定義するニーモニックを出力します。ラベルの先頭には'.' または'_' が付加されます。C--言語で public 修飾された名前に'_' が付加されます。

```
中 間 言 語 ： vmName(idx)
ニーモニック： ラベル

変換例：vmName(3)  => .a      // 名前表の 3 番目に a があった場合
```

vmTmpLab

コンパイラが自動的に生成した番号で管理されるラベルを出力します。このラベルはC--プログラムソースには存在しない名前です。整数 n で区別できるラベル ln をニーモニックに出力します。

```
中 間 言 語 ： vmTmpLab(n)
ニーモニック： ln

変換例：vmTmpLab(3)  => .L3
```

C.3.2 マシン命令

スタックマシンの命令や TaC の機械語命令に変換されるべき、中間言語命令です。

vmEntry

関数の入口処理をする命令です。idx は名前表で関数名が登録されている位置です。n は関数の中で「**同時に使用される**ローカル変数の数」です。vmEntry は、n 個のローカル変数領域を確保します。関数内でスコープが切り替わり同じ領域が複数のローカル変数で共用できる場合があるので、「**同時に使用される**ローカル変数の数」が指定されます。

```
中 間 言 語 ： vmEntry(n,idx)
ニーモニック： ラベル  ENTRY   n

変換例：vmEntry(1,3)  => .a  ENTRY  1
```

vmEntryK

カーネル関数の入口処理をする命令です。引数の意味は vmEntry と同じです。C--コンパイラに-K オプションを付けて実行した場合は、vmEntry のかわりに vmEntryK が使用されます。

```
中 間 言 語 ： vmEntryK(n,idx)
ニーモニック： ラベル  ENTRYK  n

変換例：vmEntryK(1,3) => .a  ENTRYK 1
```

vmRet

関数の出口処理をする命令です。ローカル変数を捨てて関数から戻ります。通常の関数、カーネル関数で共通に使用します。

```
中 間 言 語 ： vmRet()
ニーモニック： RET

変換例：vmRet() => RET
```

vmEntryI

interrupt 型関数の入口処理をする命令です。引数の意味は vmEntry と同じです。

```
中 間 言 語 ： vmEntryI(n,idx)
ニーモニック： ラベル  ENTRYI  n

変換例：vmEntryI(1,3) => .a  ENTRYI 1
```

RETI

interrupt 関数の出口処理をする命令です。ローカル変数を捨てて interrupt 型関数から戻ります。

```
中 間 言 語 ： vmRet()
ニーモニック： RETI
```

vmMReg

スタックから関数の返り値を取り出し、返り値用のハードウェアレジスタに移動します。

中 間 言 語 ： vmMReg()
ニーモニック： MREG

vmArg

関数を呼出す前に、関数に渡す引数を準備します。スタックから値を取り出し引数領域にコピーします。複数の引数がある場合は、最後の引数から順に処理します。

中 間 言 語 ： vmArg()
ニーモニック： ARG

以下に二つの引数を持つ関数 f を呼び出す例を示します。

```
// C-- ソース
void f(int a, int b) { ... }
void g() { f(1, 2); }

// 関数 g の中間コード
vmEntry(0,5)  // 名前表の 5 番目に g があるとする
vmLdCns(2)
vmArg()
vmLdCns(1)
vmArg()
vmCallP(2,4)  // 名前表の 4 番目に f があるとする
vmRet()

// 関数 g のニーモニック
.g
        ENTRY   0
        LDC     2
        ARG
        LDC     1
        ARG
        CALLP   2,.f
        RET
```

vmCallP

値を返さない関数を呼び出します。n は関数引数の個数、idx は名前表で関数名が登録されている位置です。

```
中 間 言 語 ： vmCallP(n,idx)
ニーモニック： CALLP n, ラベル

変換例：vmCallP(2, 4) => CALLP 2,.f
```

vmCallF

値を返す関数を呼び出します。n と idx の意味は vmCallP と同じです。vmCallF は関数の実行が終了した時、返り値をハードウェアレジスタから取り出しスタックに積みます。

```
中 間 言 語 ： vmCallF(n,idx)
ニーモニック： CALLF n, ラベル

変換例：vmCallF(2, 4) => CALLF 2,.f
```

vmJmp

無条件ジャンプ命令です。整数 n はジャンプ先ラベルの番号を表します。ラベル ln は vmTmpLab で出力される n 番目のラベルです。

```
中 間 言 語 ： vmJmp(n)
ニーモニック： JMP   ln

変換例：vmJmp(3) => JMP .L3
```

vmJT

スタックから論理値を取り出し true ならジャンプします。n、ln の意味は vmJmp と同じです。

```
中 間 言 語 ： vmJT(n)
ニーモニック： JT   ln

変換例：vmJT(3) => JT  .L3
```

vmJF

スタックから論理値を取り出し false ならジャンプします。n、ln の意味は vmJmp と同じです。

```
中 間 言 語 ： vmJF(n)
ニーモニック： JF   ln

変換例：vmJF(3) => JF  .L3
```

vmLdCns

定数 c をスタックに積みます。

```
中 間 言 語 ： vmLdCns(c)
ニーモニック： LDC   c

変換例：vmLdCns(3) => LDC  3
```

vmLdGlb

グローバル変数の値をスタックに積みます。idx は名前表で変数名が登録されている位置です。

```
中 間 言 語 ： vmLdGlb(idx)
ニーモニック： LDG   ラベル

変換例：vmLdGlb(3) => LDG .a
```

vmLdLoc

n 番目のローカル変数の値をスタックに積みます。ローカル変数の番号は 1 から始まります。

```
中 間 言 語 ： vmLdLoc(n)
ニーモニック： LDL  n

変換例：vmLdLoc(3) => LDL  3
```

vmLdPrm

現在の関数の n 番目の引数の値をスタックに積みます。引数の番号は第 1 引数から順に、1 以上の番号が割り振られます。

```
中 間 言 語 ： vmLdPrm(n)
ニーモニック： LDP  n

変換例：vmLdPrm(3) => LDP  3
```

vmLdStr

文字列リテラルの参照をスタックに積みます。整数 n はラベルの番号を表します。ラベル ln は vmTmpLab で出力される n 番目のラベルです。

```
中 間 言 語 ： vmLdStr(n)
ニーモニック： LDC  ln

変換例：vmLdStr(3) => LDC  .L3
```

vmLdLab

ラベルの値（アドレス）をスタックに積みます。idx は名前表でラベルが登録されている位置です。

```
中 間 言 語 ： vmLdLab(idx)
ニーモニック： LDC  ラベル

変換例：vmLdLab(3) => LDC  .a
```

vmLdWrd

ワード配列の要素を読み出すための命令です。まずスタックから、添字、ワード配列のアドレスの順に取り出します。次にワード配列の要素の内容をスタックに積みます。

```
中 間 言 語 ： vmLdWrd()
ニーモニック： LDW
```

vmLdByt

バイト配列の要素を読み出すための命令です。まずスタックから、添字、バイト配列のアドレスの順に取り出します。次にバイト配列の要素の内容をワードに変換してスタックに積みます。変換はバイトデータの上位に 0 のビットを付け加えることで行います。

```
中 間 言 語 ： vmLdByt()
ニーモニック： LDB
```

vmStGlb

スタックトップの値をグローバル変数にストアします。idx は名前表で変数名が登録されている位置です。スタックをポップしません。

```
中 間 言 語 ： vmStGlb(idx)
ニーモニック： STG    ラベル

変換例：vmStGlb(3) => STG .a
```

vmStLoc

スタックトップの値を n 番目のローカル変数にストアします。スタックをポップしません。

```
中 間 言 語 ： vmStLoc(n)
ニーモニック： STL    n

変換例：vmStLoc(3) => STL 3
```

vmStPrm

スタックトップの値を n 番目の引数にストアします。スタックをポップしません。

```
中 間 言 語 ： vmStPrm(n)
ニーモニック： STP  n

変換例：vmStPrm(3) => STP  3
```

vmStWrd

ワード配列の要素を書き換える命令です。まずスタックから、添字、ワード配列のアドレスの順に取り出します。次にスタックトップの値をワード配列の要素に書き込みます。後半ではスタックをポップしません。

中 間 言 語 ： vmStWrd()
ニーモニック： STW

vmStByt

バイト配列の要素を書き換える命令です。まずスタックから、添字、バイト配列のアドレスの順に取り出します。次にスタックトップの値をバイトデータに変換して配列に書き込みます。後半ではスタックをポップしません。変換はワードデータの下位 8bit を取り出すことで行います。

中 間 言 語 ： vmStByt()
ニーモニック： STB

vmNeg

まず、スタックから整数を取り出し２の補数を計算します。次に、計算結果をスタックに積みます。

中 間 言 語 ： vmNeg()
ニーモニック： NEG

vmNot

まず、スタックから論理値を取り出し否定を計算します。次に、計算結果をスタックに積みます。

中 間 言 語 ： vmNot()
ニーモニック： NOT

vmBNot

まず、スタックから整数を取り出し１の補数を計算します。次に、計算結果をスタックに積みます。

中 間 言 語 ： vmBNot()
ニーモニック： BNOT

vmChar

まず、スタックから整数を取り出し下位 8bit だけ残しマスクします。次に、計算結果をスタックに積みます。

中 間 言 語 ： vmChar()
ニーモニック： CHAR

vmBool

まず、スタックから整数を取り出し最下位ビットだけ残しマスクします。次に、計算結果をスタックに積みます。

中 間 言 語 ： vmBool()
ニーモニック： BOOL

vmAdd

まず、スタックから整数を二つ取り出し和を計算します。次に、計算結果をスタックに積みます。

中 間 言 語 ： vmAdd()
ニーモニック： ADD

vmSub

まず、スタックから整数を一つ取り出し x とします。次に、スタックから整数をもう一つ取り出し y とします。最後に、スタックに $x - y$ を積みます。

中 間 言 語 ： vmSub()
ニーモニック： SUB

vmShl

まず、スタックからシフトするビット数、シフトされるデータの順に取り出します。次に、左シフトを計算します。最後に、計算結果をスタックに積みます。

中 間 言 語 ： vmShl()
ニーモニック： SHL

vmShr

まず、スタックからシフトするビット数、シフトされるデータの順に取り出します。次に、**算術**右シフトを計算します。最後に、計算結果をスタックに積みます。

中 間 言 語 ： vmShr()
ニーモニック： SHR

vmBAnd

まず、スタックから整数を二つ取り出しビット毎の論理積を計算します。次に、計算結果をスタックに積みます。

中 間 言 語 ： vmBAnd()
ニーモニック： BAND

vmBXor

まず、スタックから整数を二つ取り出しビット毎の排他的論理和を計算します。次に、計算結果をスタックに積みます。

中 間 言 語 ： vmBXor()
ニーモニック： BXOR

vmBOr

まず、スタックから整数を二つ取り出しビット毎の論理和を計算します。次に、計算結果をスタックに積みます。

中 間 言 語 ： vmBOr()
ニーモニック： BOR

vmMul

まず、スタックから整数を二つ取り出し積を計算します。次に、計算結果をスタックに積みます。

中 間 言 語 ： vmMul()
ニーモニック： MUL

vmDiv

まず、スタックから整数を一つ取り出し x とします。次に、スタックから整数をもう一つ取り出し y とします。最後に、スタックに $x \div y$ を積みます。

中 間 言 語 ： `vmDiv()`
ニーモニック： `DIV`

vmMod

まず、スタックから整数を一つ取り出し x とします。次に、スタックから整数をもう一つ取り出し y とします。最後に、スタックに x を y で割った余りを積みます。

中 間 言 語 ： `vmMod()`
ニーモニック： `MOD`

vmGt

まず、スタックから整数を一つ取り出し x とします。次に、スタックから整数をもう一つ取り出し y とします。最後に、比較 $(x > y)$ の結果 (論理値) をスタックに積みます。

中 間 言 語 ： `vmGt()`
ニーモニック： `GT`

vmGe

まず、スタックから整数を一つ取り出し x とします。次に、スタックから整数をもう一つ取り出し y とします。最後に、比較 $(x \geq y)$ の結果 (論理値) をスタックに積みます。

中 間 言 語 ： `vmGe()`
ニーモニック： `GE`

vmLt

まず、スタックから整数を一つ取り出し x とします。次に、スタックから整数をもう一つ取り出し y とします。最後に、比較 $(x < y)$ の結果 (論理値) をスタックに積みます。

中 間 言 語 ： `vmLt()`
ニーモニック： `LT`

vmLe

まず、スタックから整数を一つ取り出し x とします。次に、スタックから整数をもう一つ取り出し y とします。最後に、比較 $(x \leq y)$ の結果 (論理値) をスタックに積みます。

中 間 言 語 ： `vmLe()`
ニーモニック： `LE`

vmEq

まず、スタックから整数を一つ取り出し x とします。次に、スタックから整数をもう一つ取り出し y とします。最後に、比較 $(x = y)$ の結果 (論理値) をスタックに積みます。

中 間 言 語 ： `vmEq()`
ニーモニック： `EQ`

vmNe

まず、スタックから整数を一つ取り出し x とします。次に、スタックから整数をもう一つ取り出し y とします。最後に、比較 $(x \neq y)$ の結果 (論理値) をスタックに積みます。

中 間 言 語 ： `vmNe()`
ニーモニック： `NE`

vmPop

スタックから値を一つ取り出し捨てます。

中 間 言 語 ： `vmPop()`
ニーモニック： `POP`

C.3.3 マクロ命令

コード生成にヒントを与えるために、ニーモニックに対応するレベルまで展開しないで、マクロ命令として中間コードを出力する場合があります。

vmBoolOR

論理 OR 式の最後で計算結果の論理値をスタックに積むマクロ命令です。整数 `n1`、`n2`、`n3` はラベルの番号を表します。論理式の途中から `true`、`false` が定まった時点でマクロ

を展開したニーモニック中の n1、n2 ラベルにジャンプしてきます。n2 が-1 の場合は短く展開されます。

```
中 間 言 語 ： vmBoolOR(n1, n2, n3)
ニーモニック： 以下のように展開されます

vmBoolOR(1, 2, 3)  | vmBoolOR(1, -1, 3)
-------------------+--------------------
      JMP   .L3    |      JMP   .L3
.L1                | .L1
      LDC   1      |      LDC   1
      JMP   .L3    | .L3
.L2                |
      LDC   0      |
.L3                |
```

vmBoolAND

論理 AND 式の最後で計算結果の論理値をスタックに積むマクロ命令です。

```
中 間 言 語 ： vmBoolAND(n1, n2, n3)
ニーモニック： 以下のように展開されます

vmBoolAND(1, 2, 3) | vmBoolAND(1, -1, 3)
-------------------+--------------------
      JMP   .L3    |      JMP   .L3
.L1                | .L1
      LDC   0      |      LDC   0
      JMP   .L3    | .L3
.L2                |
      LDC   1      |
.L3                |
```

C.3.4 擬似命令

データ生成用の疑似命令です。

vmDwName

名前へのポインタを生成します。idx は名前表で名前が登録されている位置です。

中 間 言 語 ： vmDwName(idx)
ニーモニック： DW ラベル

変換例：vmDwName(3) => DW .a

vmDwLab

vmTmpLab で出力されるラベルへのポインタを生成します。整数 n はラベルの番号を表します。ラベル ln は vmTmpLab で出力される n 番目のラベルです。

中 間 言 語 ： vmDwLab(n)
ニーモニック： DW ln

変換例：vmDwLab(3) => DW .L3

vmDwCns

ワードデータを生成します。整数 c は生成するデータの値です。

中 間 言 語 ： vmDwCns(c)
ニーモニック： DW c

変換例：vmDwCns(3) => DW 3

vmDbCns

バイトデータを生成します。整数 c は生成するデータの値です。vmDbCns はバイト配列の初期化で使用されます。

中 間 言 語 ： vmDbCns(c)
ニーモニック： DB c

変換例：vmDbCns(3) => DB 3

vmWs

ワードデータ領域（配列）を生成します。整数 n は生成するワードの数です。

中 間 言 語 ： vmWs(n)
ニーモニック： WS n

変換例：vmWs(3) => WS 3

vmBs

バイトデータ領域（配列）を生成します。整数 n は生成するバイトの数です。

```
中 間 言 語 ： vmBs(n)
ニーモニック： BS  n

変換例：vmWs(3) => BS  3
```

vmStr

文字列を生成します。整数 str は文字列です。

```
中 間 言 語 ： vmStr(s)
ニーモニック： STRING s

変換例：vmStr("hello\n") => STRING "hello\n"
```

変更履歴

2016 年 03 月 15 日 C--V.3.0.0 用作成

対応ソフトウェアのバージョン

C--	Ver.3.0.0
AS--	Ver.2.1.1
LD--	Ver.2.0.0
OBJBIN--	Ver.2.0.0
SIZE--	Ver.2.0.0
OBJEXE--	Ver.1.0.0

プログラミング言語 C--

発行年月　２０１６年３月　Ver. 3.0.0
発　　行　独立行政法人国立高等専門学校機構
徳山工業高等専門学校
情報電子工学科　重村哲至
〒 745-8585 山口県周南市学園台
sigemura@tokuyama.ac.jp