16 プリプロセッサ

一般に、cc, gcc といったC言語処理系は翻訳の前に前処理を行う。

コンパイラの作業：

(1) 前処理　(ヘッダファイルの読み込み、等を行う。)

(2) プログラムを構成する文字の列を字句、すなわち
　　コンパイルの際に意味のある最小単位
　の列に変換する。

字句には次の６種類がある。

キーワード	···	int, while, ...
識別子	···	変数名, 関数名, ...
定数	···	77, 12.3e+5, 'a', ...
文字列定数	···	"abc", ...
演算子	···	+, -, *, /, %, 関数名の次の括弧, ...
句切り記号	···	( ), { }, ;, ...

(3)
(4)
⋮
} 構文解析、翻訳コード生成、など

プリプロセッサ(前処理を行う部分):

- Cプログラム中の # で始まる行は、どういう前処理を行うかの指示をしている。[プリプロセッサ指令という。普通、1カラム目に # を置く。]

例:

```
#include <stdio.h>   ···(/usr/include/stdio.h)

#include "ファイル名"
                ···(ファイル名は普通 .h で終わる。)

#define  PI  3.14159
                ···(マクロ名には普通英大文字を使う。)
```

- プリプロセッサ指令の効力は、その指令の場所からファイルの終わりまで有効。　(但し、効力打ち消しの指令があればそこまで。)

16–1 #includeの使い方

#includeで始まる行についてのまとめ(1.3.1節)：

- #include " .h " の形の指令

  ⇨ 自分で用意したインクルードファイル
  ./ .h の中身を挿入

- #include < .h > の形の指令

  ⇨ 標準に用意されたインクルードファイル
  /usr/include/ .h の中身を挿入

- ファイルの先頭に置くのが普通。

  (⇨ 挿入指示のファイルをヘッダファイル ともいう。)

- ヘッダファイルの拡張子は習慣的に `.h`

- ヘッダファイルの中に `#include` や `#define` で始まる行があってもよい。

- 標準のヘッダファイルの中には、ライブラリ関数がどんな型のデータを引数に取りどんな型の値を返すかの情報をコンパイラに知らせるための文、などが入っている。

補足事項：

- #include " [　　.h] " の形の指令の場合、カレントディレクトリに [　　.h] という名前のファイルが無ければ、C言語処理系毎の規則に従ってファイルの探索が行なわれる。
  (UNIXの場合は/usr/include 辺り。
  ccコマンドの -Iオプションを使った指定も可。)

- 複数のファイルにまたがる大きなプログラムを構築する場合、

```
#include <stdio.h>
    ⋮
```

マクロ定義の列
構造体定義の列
新しいデータ型定義の列
関数プロトタイプの列

というヘッダファイルを作り、各ファイルの先頭にこのヘッダファイルをincludeする指令を入れておけば、同じ定義／宣言をあちこちのファイルの先頭で繰り返す手間が省ける。

16–2 #defineの使い方

例16. 1 (#define行の使用例)　マクロ定義を使った例としては、

例題4.1(三角関数の表),
例題4.5(quicksort),
例題8.1(Napier数eの1000桁計算)

を挙げることが出来る。

```
 6  #define PI (3.1415926535897932) /*円周率*/
```
…(例題4.1,三角関数の表)

```
 9  #define  SIZE   100
10  #define  WIDTH  10
11  #define  TRUE    1
```
…(例題4.5,quicksort)

```
13  #define  LIMIT   7
```
…(例題8.1, eの1000桁計算)

—

#defineで始まる行についてのまとめ(1.3.1節)：

- マクロ定義という。
- これを用いれば、プログラムのパラメータとなる定数、物理定数などに記号の名前(マクロ名 または 記号定数という)を付け、以降のプログラム内で自由に使うことが出来る。
- 習慣的に、マクロ名には英大文字列を使う。
- マクロ定義によってパラメータ付きの任意の文字列に名前を付けることが出来る。 例えば、

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

但し、この場合は max(i++,j++) とすると駄目。

- マクロを定義する場合、マクロ名の右側の置換テキストは全体を丸括弧で囲むのが無難。 何故なら、例えば

```
#define square(x) (x)*(x)
```

とマクロ定義した場合は、

```
4/square(2)  ⇨  4/(2)*(2)
```

と展開されてしまう。

- パラメータ付きマクロを定義する場合、マクロ名の右側の置換テキストにおいては各パラメータを丸括弧で囲むのが無難。 何故なら、例えば

```
#define square(x) (x*x)
```

とマクロ定義した場合は、

```
square( z+1 )  ⇨  ( z+1*z+1 )
```

と展開されてしまう。

2つの形式：

#define マクロ名 文字列

#define マクロ名(仮引数名,...,仮引数名) 文字列

↑
間に空白なし

補足事項：

- 行の終りに\を置けば次の行に続けることが出来る。
- マクロ定義をうまく使えば、プログラムの明解さ、可搬性が向上する。

特殊な定数,
プログラムのパラメータ } ⇨ 引数なしのマクロ定義

マクロ定義の解消

次の様に書く。

#undef マクロ名

16–3 引数付きマクロの使い方

例16. 2 (引数付きマクロの使用例)

引数付きマクロ定義を使った例としては、

例題8.2(○月○日のカレンダー),

例題11.3(N次元ベクトル空間の世界)

を挙げることが出来る。

```
4   #define   DUMMY      0
5   #define   min(A,B)   ((A) < (B) ? (A) : (B))
```

···(例題8.2,○年○月のカレンダー)

```
#define  N  3
#define  Print(title, vector)          \
    printf("%s\n", title);             \
    printf("   (%7.3f ", vector[0]);   \
    for (i=1; i<N; ++i)                \
      printf("%7.3f ", vector[i]);     \
    printf(")\n")
```

…(例題11.3,N次元ベクトル空間の世界)

⇨

引数付きマクロは、簡単な式だけでなく、繰り返しや条件分岐等も含む作業手順を1つにまとめ上げるのに有用である。

補足(関数の場合と比較)：

オブジェクト コードが幾分大きくなる が、
関数 パラメータの引渡しは不要 である。

形式：

#define マクロ名 (仮引数名 , . . . , 仮引数名) 文字列

機能：

- 以降に現われる マクロ名 (仮引数名 , . . . , 仮引数名) というパターンを全て 文字列 に置き換える。その際、 文字列 の中に現われる引数名は,マクロ呼出し時の対応する実引数でそれぞれ置き換える。

  ⇨ 関数の代わりに使えば、計算効率が良い。

- 文字列置き換え後の構文が正しいかどうかのチェックはしない。

- 文字列 中に現われる仮引数名 の前に # が(1個だけ)付いていると、対応する実引数を2重引用符で囲んだものに置き換わる。

- 文字列 中に ## があると、仮引数が対応する実引数に置き換えられた後で、## とその両側の空白が除去される。

例16. 3 (良い例)

- `#define SQ(x) ((x)*(x))`
- `#define Min(x,y) (((x) < (y)) ? (x) : (y))`

例16. 4 (失敗例)

- `#define SQ(x) (x*x)`
  (理由: `SQ(a+b)` が `(a+b*a+b)` に展開される。)
- `#define SQ(x) (x)*(x)`
  (理由: `4/SQ(2)` が `4/(2)*(2)` に展開される。)
- `#define␣␣SQ␣(x)␣␣((x)*(x))`
  (理由: `SQ(7)` が `(x)␣␣((x)*(x))(7)` に展開される。)
- `#define SQ(x) ((x)*(x));`
  (理由: `SQ(2)+1` が `((2)*(2));+1` に展開される。)

16–4 演算子#と##

例16. 5 (#演算子)　デバッグ用に

```
#define Dprint(expr) printf(#expr " = %g\n", expr)
```

と定義すれば、例えば、

```
Dprint(x/y);
```

は次の様に置き換わる。

```
printf("x/y" " = %g\n", x/y)
```

すなわち

```
printf("x/y = %g\n", x/y)
```

例16. 6 (##演算子)

```
#define X(i) x ## i
```

と定義すれば、例えば、

```
X(1)=X(2)=X(3);
```

は次の様に置き換わる。

```
x1=x2=x3;
```

16–5 条件付きコンパイル

目的：

- プログラム開発を容易にする(例えば、デバッグ用のコードを使うかどうかの切り替えを行なう)ため。
- 容易に移植できるコードを書くため。

補足：
アーキテクチャに依存したコードの切替えを容易に行えるということ。

使い方：

- プログラムの中の

```
#if 定数式
    ⋮
#endif
```

の部分は 定数式 の値がゼロ以外(真) の時だけコンパイルされる。

- プログラムの中の

```
#ifdef マクロ名
   ⋮
#endif
```

または

```
#if defined(マクロ名)
   ⋮
#endif
```

の部分は #ifdef の場所で マクロ名 の値が定義されている時だけコンパイルされる。

- プログラムの中の

```
#ifndef マクロ名
   ⋮
#endif
```

または

```
#if !defined(マクロ名)
   ⋮
#endif
```

の部分は #ifndef の場所で マクロ名 の値が定義されていない時だけコンパイルされる。

例16. 7 (デバッグ用コードをコンパイルするかどうか、の切り替え)

変数 a の途中の値を観察することがプログラムの動作をチェックする上で有用な場合、a の値を覗いてみたい場所に

```
#if DEBUG
 printf("debug: a = %d\n", a);
#endif
```

というコードを埋め込んでおけば、プログラムの先頭で

```
#define DEBUG 1
```

とするか

```
#define DEBUG 0
```

とするかの切り替えだけで、変数 a の途中の値を観察するかどうかの切り替えを行なうことが出来る。

その他：

- if-else 文に似た制御構造も用意されている。

```
#if 定数式    /* #ifdef や #ifndef も可 */
  .........
#elif 定数式
  .........
#elif 定数式
  .........
  .........
#else
  .........
#endif
```

16–6 既定義のマクロ

既定義マクロ名	値
__DATE__	前処理時の日付を表す文字列。
__FILE__	ソースファイルのファイル名から成る文字列。
__LINE__	現在の行番号を表す整数。
__STDC__	ANSI Cコンパイラであれば、ゼロでない整数。
__TIME__	前処理時の時間を表す文字列。

16–7 自習 assert() マクロ

assert() マクロ：

- 標準ヘッダファイル <assert.h> の中で定義されている引数付きマクロ。

- プログラムの中の

  assert(式);

  というマクロ呼出しは

  式 が偽 (0) ならメッセージを出して強制終了する

  というコードに置き換えられる。

assert() マクロの定義の様子

```
[motoki@x205a]$  cat /usr/include/assert.h
      (中略)
#ifdef NDEBUG
# define assert(expr)  ((void) 0)
      (中略)
#else /* Not NDEBUG. */
/* This prints an "Assertion failed" message
    and aborts. */
extern void __assert_fail __P ((            __const char
*__assertion,
           __const char *__file,
           unsigned int __line,
           __const char *__function))
   __attribute__ ((__noreturn__));
      (中略)
# define assert(expr)                                          \
((void) ((expr) ? 0 :                                          \
        (__assert_fail (__STRING(expr),                        \
                        __FILE__, __LINE__,__ASSERT_FUNCTION), 0)))
      (中略)
#endif /* NDEBUG. */
```

16–8 自習 #error と #pragma, #line

#error 指令：

- 形式は

  #error " エラーメッセージ "

- (前処理時に) #error に出会うとコンパイル時にエラーと判断し、#errorに続く文字列を画面に出力させるための指令。
- この指令が実行されるとコンパイル処理は中断され、実行コードは生成されない。

例16. 8 (#error指令)

```
[motoki@x205a]$  nl preprocessor-error-directive.c
     1 #include <stdio.h>

     2 int main(void)
     3 {
     4   printf("ファイル \"%s\" のコンパイル開始"
                                 " (date: %s, time: %s)\n",
     5           __FILE__, __DATE__, __TIME__);

     6 #error ----------Check----------

     7   printf("%d\n", 777);/*コンパイルされない*/
     8
     9   return 0;
    10 }
[motoki@x205a]$  gcc preprocessor-error-directive.c
preprocessor-error-directive.c:8: #error ----------Check-------
[motoki@x205a]$  ./a.out
bash: ./a.out: そのようなファイルやディレクトリはありません
[motoki@x205a]$
```

#line指令：

- 形式は

  #line 行番号 " ファイル名 "

- コンパイル中にCコンパイラが保持する
  - 行番号 と ···(マクロ __LINE__ で表される)
  - ソースファイル名 ···(マクロ __FILE__ で表される)

  の情報を強制的に変更するための指令。

- 行番号 は次の行に設定する行番号を表す。

例16. 9 (#line指令)

```
[motoki@x205a]$  nl preprocessor-line-directive.c
     1 #include <stdio.h>

     2 int main(void)
     3 {
```

```
 4   printf("ファイル \"%s\" の %d 行目を"
                                "コンパイル中\n"
 5       "    (date: %s, time: %s)\n",
 6       __FILE__,__LINE__,__DATE__,__TIME__);     7行目
                                                   8行目
 7 #line 100 "myprog.c"                            9行目
                                                 100行目
 8   printf("ファイル \"%s\" の %d 行目を"        101行目
                                "コンパイル中\n"
 9       "    (date: %s, time: %s)\n",            102行目
10       __FILE__,__LINE__,__DATE__,__TIME__);   103行目
11
12   return 0;
13 }
[motoki@x205a]$  gcc preprocessor-line-directive.c
[motoki@x205a]$  ./a.out
ファイル "preprocessor-line-directive.c" の 7 行目をコンパイル
中
    (date: Mar 14 2003, time: 17:06:06)
ファイル "myprog.c" の 103 行目をコンパイル中
    (date: Mar 14 2003, time: 17:06:06)
[motoki@x205a]$
```

#pragma指令：

- 形式は

  #pragma 命令

- 個々のコンパイラに応じた処理をするための指令。
- コンパイラが認識できない#pragma指令は無視される。

例16. 10 (#pragma指令) **OpenMP**では、**#pragma指令**を埋めこむことによって**並列処理の指示**を行う。

```
[motoki@x205a]$  nl preprocessor-openMP.c
     1 // #pragma指令の使用例(OpenMP)
     2 #include <omp.h>
     3 #include <stdio.h>

     4 int main(void)
     5 {
```

```
 6   int i, a[10]=0;

 7   printf(" a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8]
 8   #pragma omp parallel
 9   {
10     #pragma omp for
11     for (i=0; i<10; i++)
12     {
13       a[i] = 80 + i;
14       printf("%4d %4d %4d %4d %4d %4d %4d %4d %4d %4d"
15              " ...(i=%2d, thread_num=%d)\n",
16              a[0], a[1], a[2], a[3], a[4], a[5], a[6],
17              i, omp_get_thread_num());
18       fflush(stdout);   //起こった順に正確に表示するため

19     }
20   }
```

```
21   return 0;
22 }
[motoki@x205a]$  gcc -fopenmp preprocessor-openMP.c
[motoki@x205a]$  ./a.out
 a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]
  80    0    0    0    0   85    0    0    0    0 ...(i= 5, th
  80    0    0    0    0    0    0    0    0    0 ...(i= 0, th
  80   81    0    0    0   85    0    0    0    0 ...(i= 1, th
  80   81   82    0    0   85    0    0    0    0 ...(i= 2, th
  80   81   82   83    0   85    0    0    0    0 ...(i= 3, th
  80   81   82   83   84   85    0    0    0    0 ...(i= 4, th
  80   81   82   83   84   85   86    0    0    0 ...(i= 6, th
  80   81   82   83   84   85   86   87    0    0 ...(i= 7, th
  80   81   82   83   84   85   86   87   88    0 ...(i= 8, th
  80   81   82   83   84   85   86   87   88   89 ...(i= 9, th
[motoki@x205a]$
```

16–9 自習 引数付きマクロと同じ名前の標準ライ

- 標準ライブラリ関数と同じ名前の引数付きマクロが用意されていることもある。

- 標準ライブラリ関数の方を呼び出したければ、関数名を丸括弧で囲んで書く。例えば、

```
(isalpha)(c)
```

補足：
関数の引数を囲む丸括弧も演算子なので、これで関数を呼び出せる。

マクロを避けて関数を使うと副作用の影響を考えなくて済む。