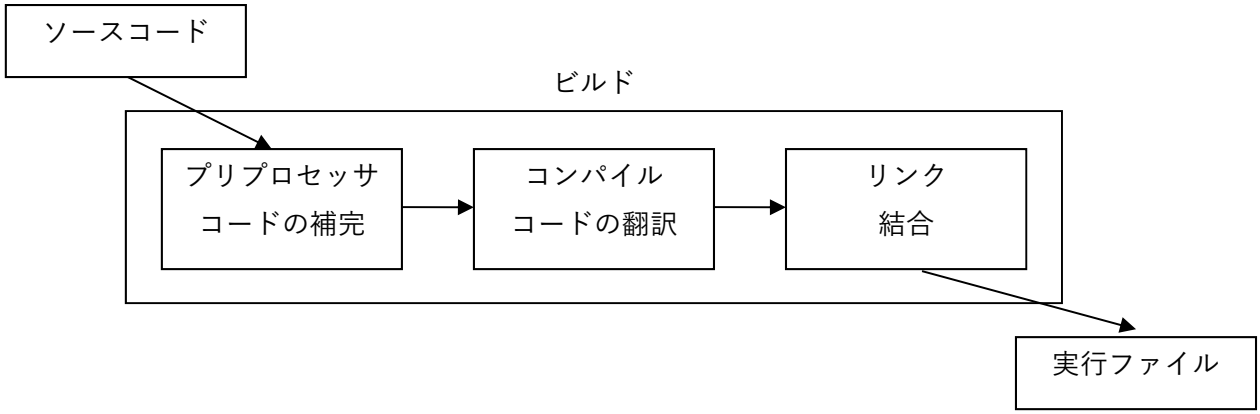
C 言語 020 プリプロセッサ

【プリプロセッサ】

プリプロセッサは「前処理」とも言い、コンパイル前に行う作業のことで、主にソースコードの「補完作業」を行うものです。

＜ビルドの流れ＞



プリプロセッサには下記の種類があります。

1. インクルード
2. マクロ定義
3. 条件付きコンパイル

【インクルード】

インクルードは指定したファイルのソースコードを組み込む処理です。

header.h

```
struct personal {
    char name[10];
    int  age;
};

void func(void);
```

sauce.c

```
#include "header.h"
int main(void) {
    func();
```

補完後のコード

```
struct personal {
    char name[10];
    int  age;
};

void func(void);

int main(void) {
    func();
```

sauce.c の「#include "header.h"」の部分に header.h のソースコードが組み込まれます。
インクルードはヘッダーファイルだけでなくソースファイル（拡張子 c）も可能ですが、そのような使い方はしません。

【マクロ定義】

「マクロ定義」は値や処理を別名で置換する処理です。

＜マクロ定義＞

```
#define マクロ名 値・処理
```

マクロ名は大文字で定義するのが一般的です。

＜例文 1＞

```
#include <stdio.h>
#define VALUE 10              /* マクロ名「VALUE」を「10」と定義 */
int main(void) {
    printf("%d¥n", VALUE);    /*「VALUE」は「10」に置換される */
    return 0;
}
```

＜例文 1 の補完後のコード＞

```
int main(void) {
    printf("%d¥n", 10);
    return 0;
}
```

式もマクロ定義できます。

＜例文 2＞

```
#include <stdio.h>
#define EXPR x + y          /*マクロ名「EXPR」を「x + y」に定義 */
int main(void) {
    int x = 10;
    int y = 20;
    printf("%d¥n", EXPR);    /*「EXPR」は「x + y」に置換される */
    return 0;
}
```

＜例文 2 の補完後のコード＞

```
int main(void) {
    int x = 10;
    int y = 20;
    printf("%d¥n", x + y);
    return 0;
}
```

【引数付きマクロ定義】

マクロに引数を付けて関数のように使用できます。

＜引数付きマクロ定義＞

```
#define マクロ名(引数) 処理
```

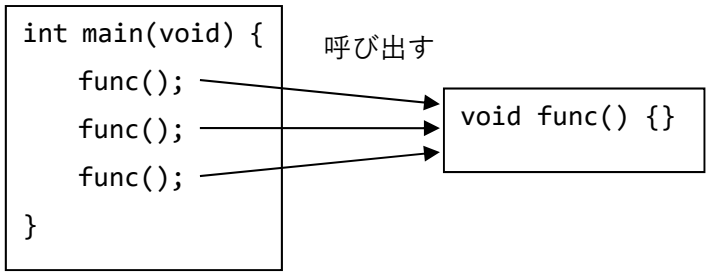
＜例文＞

```
#include <stdio.h>

/* マクロを関数のように使用する。¥はマクロ内での改行を示す */
#define TRIANGLE(b, h) ¥
    double a = b * h / 2.0; ¥
    printf("三角形の面積は%.2lf です", a);

int main(void) {
    double b = 10;
    double h = 20;
    TRIANGLE(b, h);    /* マクロ「TRIANGLE」に引数が渡され置換される */
    return 0;
}
```

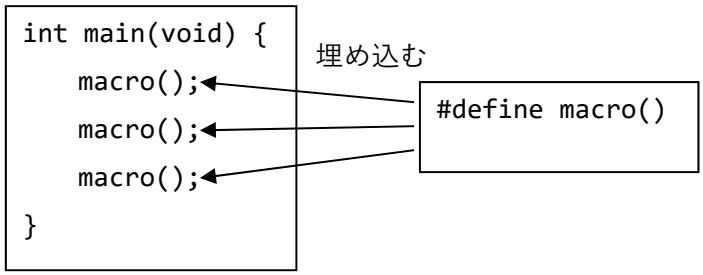
＜実行結果＞

三角形の面積は 100.00 です

関数は使用時にプログラム領域から呼び出されるため、処理に時間がかかります。



引数付きマクロ定義は呼び出し側にコードを埋め込むので処理時間が速くなります。
しかし、その分、呼び出し側のコードが肥大になり、プログラムサイズが大きくなってしまいます。



【条件付きコンパイル】

マクロ定義を使用して、コンパイルするソースコードとコンパイルしないソースコードを区別する処理です。

＜書き方 1＞

```
#if マクロ名
    コード    /* マクロが真だった場合はこのコードがコンパイルされる */
#else
    コード    /* マクロが偽だった場合はこのコードがコンパイルされる */
#endif
```

#else は必ずしも指定する必要はありません。

＜例文 1＞

```
#include <stdio.h>
#define MACRO 1    /* 真は 0 以外、偽は 0 */
int main(void) {

#if MACRO
    printf("MACRO は真");
#else
    printf("MACRO は偽");
#endif

    return 0;
}
```

＜コンパイル後のプログラムの実行結果＞

MACRO は真

同ソースファイル内でもコンパイルする部分としない部分を切り分けることができ、プログラムサイズを小さくすることができます。

if 文はプログラムの中で分岐を行いますが、条件付きコンパイルはコンパイル前に行うため、コンパイル後は分岐することができません。

条件付きコンパイルは、Windows 用と UNIX 用でコンパイルするコードを分けたり、デバック用（変数の値を printf したり）と本番用で分けたりするのに使用されます。

＜書き方 2＞

```
#ifdef マクロ A（#if defined でも可）
    コード    /* マクロ A が定義されている場合はこのコードがコンパイルされる */
#elif defined マクロ B
    コード    /* マクロ B が定義されている場合はこのコードがコンパイルされる */
#else
    コード    /* マクロ A もマクロ B も定義されていない場合はこのコードがコンパイルされる */
#endif
```

#elif defined、#else は必ずしも指定する必要はありません。

＜例文 2＞

```
#include <stdio.h>

/* WIN と定義する */
#define WIN

int main(void) {

#ifdef WIN            /* WIN が定義されている場合 */
    printf("Windows 用にコンパイルします");
#elif defined UNIX    /* UNIX が定義されている場合 */
    printf("UNIX 用にコンパイルします");
#else                 /* どっちも定義されていない場合 */
    printf("汎用にコンパイルします");
#endif

    return 0;
}
```

＜コンパイル後のプログラムの実行結果＞

Windows 用にコンパイルします

＜コンパイル時にマクロを定義する＞
ソースコード内でマクロを定義するのではなく、コンパイル時にマクロを定義できます。
コマンドから「-D マクロ名」で指定します。

gcc -o *** -D マクロ名 ***.c

```
#include <stdio.h>
#include <stdlib.h>

/* メイン関数に引数を指定する */
int main(int argc, char* argv[]) {

    /* コマンドラインから入力された金額 */
    double money = atof(argv[1]);

#ifdef JPN    /* -DJPN で日本の消費税で計算 */
    double rate = 1.08;
    int tax = money * rate;
    printf("日本での消費税は%d 円です¥n", tax);
#endif

#ifdef NY     /* -DNY でニューヨークの消費税で計算 */
    double rate = 1.08875;
    double tax = money * rate;
    printf("tax is $%.2f in NY¥n", tax);
#endif

    return 0;
}
```

コンパイル時に「-DJPN」を指定すれば日本用、「-DNY」を指定すればニューヨーク用になります。

【デバック】

＜assert＞

アサート（assert）はデバック作業で使用される機能です。

「assert」は assert.h にマクロ定義されています。引数に条件式を指定します。プログラム実行時に、その条件式が「偽」であれば、プログラムを中断するようになっています。

＜使い方＞

```
assert( 条件式 )
```

＜例文＞

```
1  #include <stdio.h>
2  #include <assert.h>      /* アサートに関するヘッダーファイル */
3
4  int main() {
5      int* p = NULL;
6      assert(p != NULL);    /* この条件式は偽にる */
7      printf("p=%p", p);
8      return 0;
9  }
```

6 行目までに、変数 p が NULL だとエラーにするように作られています。

コンパイルして実行すると 6 行目で assert が実行され、プログラムが中断されます。

```
C:¥Users¥eitarou¥Desktop>assert
Assertion failed!

Program: C:¥Users¥eitarou¥Desktop¥assert.exe
File: assert.c, Line 6

Expression: p != NULL

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

＜assert マクロを OFF にする＞

assert マクロを OFF にしたい場合はコンパイル時に「-DNDEBUG」をマクロ定義します。

これは「assert.h」にて「#ifdef」が記述されています。

＜デバック用のソースを組み込む＞

「 __FILE__ 」（マクロ名の前後はアンダーバー2 個）はソースファイル名、「 __LINE__ 」はソースコードの行数を取得するためのマクロ定義です。

ファイル名「debug.c」

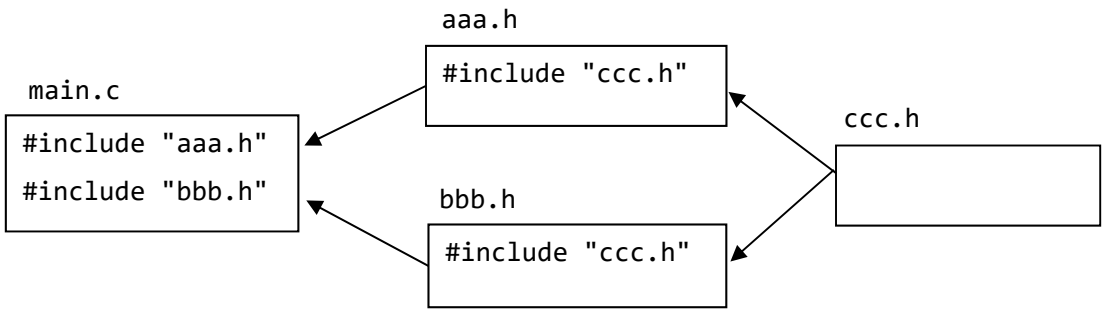
```
#include <stdio.h>

/*「-DDEBUG」でデバック情報を表示する */
#ifdef DEBUG
#define DEBUG_SHOW(val) printf("DEBUG:%s(%d):%d¥n", __FILE__, __LINE__, val)
#else
#define DEBUG_SHOW(val)
#endif

int main(void) {
    int i;
    for(i = 0;i < 10;i++){
        DEBUG_SHOW(i);    /* この行は 14 行目 */
    }
    printf("i=%d", i);
    return 0;
}
```

コンパイル時に「-DDEBUG」を指定すればデバック情報を出力します。

＜-DDEBUGでコンパイルした後の実行結果＞

```
DEBUG:debug.c(14):0
DEBUG:debug.c(14):1
DEBUG:debug.c(14):2
DEBUG:debug.c(14):3
DEBUG:debug.c(14):4
DEBUG:debug.c(14):5
DEBUG:debug.c(14):6
DEBUG:debug.c(14):7
DEBUG:debug.c(14):8
DEBUG:debug.c(14):9
i=10
```

【多重インクルード】



上記の図は 4 つのファイルのインクルード関係を表しています。
「main.c」は「aaa.h」と「bbb.h」をインクルードしており、「aaa.h」と「bbb.h」は「ccc.h」をインクルードしています。
「main.c」をコンパイルすると「ccc.h」を 2 回インクルードすることになりエラーになります。

＜多重インクルード防止の記述＞
多重インクルードを防止するには「#ifndef」を利用します。「#ifndef」は「#ifdef」と逆の効果を持っており「マクロが定義されていなかったらコンパイルする」処理を行います。

＜ccc.h に下記のコードを記述＞

```
#ifndef __CCC_H__    /* マクロ名「 __CCC_H__ 」が定義されていなかったらコンパイルする */
#define __CCC_H__    /* マクロ名「 __CCC_H__ 」を定義する */
    ソースコード
#endif
```

この記述により「ccc.h」は 1 回目のインクルードでマクロが定義され、ソースコードが組み込まれますが、2 回目のインクルードでは #ifndef により組み込まれず、多重インクルードを防止することができます。
汎用的な関数のヘッダーファイル等は複数のファイルからインクルードされる可能性が高いので多重インクルード防止を行うといいです。
また、マクロ名は「 __CCC_H__ 」のようにファイル名を大文字で、前後をアンダーバー2 個で括るのが一般的です。