# ソースコードの色

予約語 ：int,if,typeof など

Enum ：など

Class ：UserControl など

コメント：

文字列：String

# プログラム基本

## エントリーポイント

int main(int argc, char \*argv[]){

{

}

## 複数ファイル

C言語で複数ファイルに分ける方法。

ファイル構成

/*multi\_files* ただの格納フォルダで，名前は何でも良い。

├ main.c エントリ・ポイントを含むファイルは慣例的にmain.c

├ hello.c 追加するソースファイル本体。

└ hello.h 追加するソースファイルのヘッダ。目録みたいな感じ。

main.c

// mainの方では stdio.h のincludeが必要ない。（むしろしない方が良い）

#include "hello.h" // <>　でなく，""　である点に注意。

int main(int argc, char \*argv[]){

sayHello();

}

hello.h

/\* ここにメソッドの説明書きがある事が多い \*/

int sayHello(); // メソッドの中身を書かない。

hello.c

#include "hello.h" // 真っ先に includeするべき　という声が多い

#include <stdio.h> // printf

int sayHello(){

printf("hello world\n");

}

コンパイル

gcc -o *hello* *main*.c *hello*.c # 必要ファイルをすべて記述。（.cファイルの名前だけでよい）

./hello # コンパイルした実行ファイルを起動。

複数ファイル時の注意点

・別ファイルでも使う変数にはexternをつける。

個人的にはexternは極力使うべきでないと考える。面倒でも，引数や戻り値などで渡すべき。

・複数ファイルで同じ名前の変数やメソッドを宣言しない。（スコープが異なれば宣言できる）

## 予約語

※C言語特有のもののみ。他の言語をやっていれば分かるようなものは排除した。

true = 1 らしい。e.g. while(1){} // ⇒ 無限ループになる。

auto

struct 構造体を定義する。メンバ変数がそれぞれ別々のメモリ領域を使用する。

switch ifを連続で使ったような条件分岐。様々なプログラム言語で使用されるので説明不要かと。

register

typedef 型を定義する。

extern グローバル変数を定義する。

union 共用体を定義する。メンバ変数が共通のメモリ領域を共用する。意味ある？

continue

default

goto

sizeof

volatile 変更の可能性があり，ROMでは無くRAMに置く。割込み中にアクセスする変数など。

inline

restrict

far far メモリに置かれることを示すポインタ。メモリ上の実際のアドレスを示す。

near near メモリに置かれることを示すポインタ。特定のメモリ番地からの相対アドレス。

※farとnear はMS-DOSの仕様。

## ハンガリアン命名規則

特にC++で好んで用いられた命名規則。近年の言語ではむしろ非推奨されている。

過去のソースコードを読む際の資料として残す。

### プリフィックス（接頭辞）

|  |  |  |  |
| --- | --- | --- | --- |
| 単語 | 意味 | 例 | 備考 |
| a | 配列 |  |  |
| b 　または f | 論理型 | bDirtyFlag |  |
| by | バイト型 (符号なし1バイト整数) | byGrayLevel |  |
| c\_ | 定数 (const) | c\_nBufferSize |  |
| ch | 文字型 | chSeparator |  |
| C | クラス | CHoge |  |
| d　またはdb | double（倍精度浮動小数点型） |  |  |
| dw | DWORD　unsigned long | dwSize |  |
| e | 配列の要素 |  |  |
| f　またはfp | float（単精度浮動小数点型） | fPrice |  |
| fn | 関数ポインタ型 | fnCallback |  |
| g\_ | グローバル変数 | g\_iErrorCode |  |
| h | ハンドル型 | hThread |  |
| hwnd または h | ウィンドウハンドル型 (Windows OS) | hMainWindow |  |
| i 　またはn | int, integer（4byte数値） | iPower | バイト数は環境による |
| in | クラスのインスタンス変数 |  |  |
| l | 長整数 (long) | lDate |  |
| m\_ | クラスのメンバ変数 | m\_nLength |  |
| p 　または lp | ポインタ型 | lpDirectSound |  |
| s | 文字列型 | sPlayerName |  |
| sh | short（2byte数値） |  |  |
| sz | NULLで終わる文字列型 | szFileName |  |
| s\_ | 静的変数 (static) | s\_pLookupTable |  |
| tag | 構造体タグ | tagRECT |  |
| u | 符号なし整数 | uiCount |  |
| ush | unsigned short |  |  |
| w | ワード型 (WORD)　unsigned int | wLanguageCode |  |
|  |  |  |  |

参考：<http://kikky.net/pc/hungarian.html>

参考：<https://ja.wikipedia.org/wiki/ハンガリアン記法>

## プレフィックス

0b 2進数を表す

0 8進数を表す

0x 16進数を表す

## サフィックス

変数名の先頭に暗黙の了解でつける文字の事。（近年の言語ではむしろ非推奨されている）

U int

F 浮動小数点(float)

L ロング

UL 符号無し(unsigned)ロング

## メタ文字

\t タブ

\n 改行

## データ型

先に注意点：

・標準ではC言語にはbool型は存在しない。

（C99からの新機能として存在してはいるらしいがいずれにせよ #define true 1 といった形）

### 基本データ型

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | 範囲 | suffix | .NET |
| char | 1バイト符号付整数。主に英数字を一字記憶させる用途に使用する。 | -128～127 |  | char  sbyte |
| short | 2バイト符号付整数。 | -32,768～32,767 | なし | Int16  short |
| int | int型は環境により記憶容量が変化する。32bit環境 4バイト整数  64bit 環境 8バイト整数　(Windows) |  | なし | 該当なし  強いて言うならdecimal |
| long | 4バイト符号付整数。-の値を記憶できる。 | 2,147,483,64～2,147,483,647 | L | Int32  int |
| long long | 8バイト符号付整数。 | 9,223,372,036,854,775,80～9,223,372,036,854,775,807 |  | Int64  long |
| float | 4バイト単精度浮動小数点実数。 |  | F | Single  float |
| double | 8バイト倍精度浮動小数点実数。 |  | 10.0 など | double |
| signed | 符号あり。通常signed は省略する。 |  |  | 該当なし |
| unsigned | 符号無し。 unsigned short などとして使う。 | U ；unsigned short  UL ；unsigned long | | 該当なし |

### typedef型定義

だいたい　<stdint.h>　で以下の様な定義がなされている

typedef unsigned char　uchar; //

typedef unsigned short ushort;

typedef unsigned int uint;

typedef unsigned long ulong;

typedef signed char \_SBYTE;

typedef unsigned char \_UBYTE;

typedef signed short \_SWORD;

typedef unsigned short \_UWORD;

typedef signed int int;

typedef unsigned int uint;

typedef signed long \_SDWORD;

typedef unsigned long \_UDWORD;

typedef signed long long \_SQDWORD;

typedef unsigned long long \_UQDWORD;

※但し，個人的にはtypedef は使用しない方がはるかに良いと考える。

補足：C++くらいまでは「byte」というデータ型が存在しなかった為，\_UBYTE, uint8\_tなどのtypedefで作られた型を使っていたようだが，これらは全て unsigned char。

## 配列

配列の長さ（要素数）をみる

#define ARRAY\_SIZE(a) (sizeof(X)/(sizeof(X[0])))

補足：sizeofはデータ型，もしくは配列変数に使われる

sizeof(short)); // > 2と出力される

short sh\_array[]={1,2,3};

sizeof(short)); // > 6と出力される 　※書き間違いでは2021-09-07

配列を関数の引数にする

void someMethod1(int array[3]){ // 要素数の明示

}

void someMethod2(int array[]){ // 要素数を明示しない

}

void someMethod3(int \*array){ // ポインタを使う

}

void someMethod3(void \*array){ // いろんな型を受け取る事ができる

} //　.NETのobject[] みたいな感じ

配列に関する注意点

関数で宣言したローカル変数を関数の戻り値にしてはいけない。（コンパイル時に警告が出る）

# プログラム応用

## ポインタ

通常の変数は新たにメモリ番地を確保するのに対し，ポインタは同じメモリアドレスを共有する。

Windowsなんかのファイル操作で例えると分かりやすいかも。変数を作成するのはファイルのコピー，または新規作成で，ポインタはそれらファイルへのショートカットの作成。

### 基本

変数の宣言時に　\*　を付ける事でポインタ変数が生成できる。

この変数は基本的には「メモリアドレス」しか受け付けない。

また，通常の変数に& をつける事でメモリアドレスの転送ができる。

例：

int ival = 10; // int 型の変数を宣言＋値を代入。

int \*intptr; //　intポインタを生成。メモリアドレスか，配列のみ受付ける変数。

intptr = &ival; //　& を使う事で，対象の変数のメモリアドレスを転送する。

\*intptr = 2; // \* を付けるとポインタ変数を普通の変数として使える。

printf(“%d\n”, ival1); // 結果　> 2

まとめ

|  |  |  |
| --- | --- | --- |
|  | intptr　　ポインタとして働く | **\***intptr　　変数として働く |
| **&**ival  メモリアドレス | intptr = &ival;  ivalのメモリアドレスをintptrに格納。  intptrとivalは同じメモリアドレスを参照。ivalの値を変えるとintptrも変わる。 | \*intptr = &ival;  変数にメモリアドレスを格納しようとしている。コンパイル警告。値の代入もされない為、　Segmentation fault　になる。 |
| ival  値 | intptr = ival;  ポインタに値を代入しようとしている。  コンパイル警告。値の代入もされない為、　Segmentation fault　になる。 | \*intptr = ival;  intptrを変数として扱い，ivalの間を転送する。  メモリアドレスの参照はされない。 |

### 何が便利か

・「参照型」の変数として扱える

通常method（関数）は戻り値を１つしか返さない。ただ，場合によっては処理の結果を2つ以上の変数で返して欲しい場合がある。

・型変換を行う

・コールバックで使う

その他

int\* intptr; // 型と変数名の間でアスタリスク(\*)を使えば良く，

int \* intptr; // この良い。

printf(“the address of ival:%p, &ival”); // アドレスを表示

int iarray[]={10,20};// 単純な配列

\*intptr = iarray[1]; // こうする事で配列の任意の要素を先頭にできる。

### アロー演算子

instance->intMember // このようなアクセス方法の事。ポインタ変数のメンバにアクセスする。

例）

TestClass test;

test.int\_member = 1; // 通常のインスタンスには . (ドット) を使ってアクセス

TestClass \* p\_test = &test;

p\_test->int\_member = 2; // ポインタ変数には，-> (アロー) を使ってアクセス。

※C言語においては，**変数（実態）を使っているか，ポインタを使っているかが非常に重要**。

## 割込み

### 基本

### 注意点

（別ドキュメント　その他マイコン　で編集中）

用語

ISR；Interrupt Service Routine

割込みハンドラ。割込みが発生した時に対処するメソッドなど。

IRAM；Instruction RAM

Data RAMの内，主にプログラムなどを保持するデータの領域。

エッジ割込み 立ち上がり割込み

## プリディレクティブ

コンパイラに渡す前に処理されるコード類の事。簡単に言うと文字列の置換えを行う。

### #define

先に注意点。#define 関連のバグ話は尽きる事がない。**できるだけ使用を控えた方が良い**と思う。

文字列の置換。　#define の使用パターンは３つある

１．定数的使い方

構文：define 文字列１ 文字列２

#define *\_\_MAX\_\_*　100 //

ただしここで行っているのはあくまでも「**文字列の置換**」であってconst とは異なる。

例）

#define IS\_NOT\_ZERO　!=0 //

int ival = 10;

if (ival IS\_NOT\_ZERO) // const ではこんな事はできない

// 処理

⇒ static const 変数を用いるべきと言う声もある。

２．実行つきコンパイル #if

定義名は大文字アルファベットを１組の\_（アンダーライン）２つで囲うのが慣例らしい

#define *\_\_IS\_TEST\_\_* 1 // 1 が true である事を利用している

#if *\_\_IS\_TEST\_\_*\_

// 処理

#endif

#define *\_\_DEFINEDNAGE\_\_* // こちらは定義してあるかどうかだけで条件判断

#ifdef *\_\_DEFINEDNAGE\_\_*\_

// 処理

#endif

３．関数マクロ

#define MULTI( x, y ) x \* y // 引数を取る（小文字で書くと引数？）

printf( MULTI(1, 2) );

組み込みマクロ

\_\_FILE\_\_ // 展開された場所のファイル名の文字列に置換されます

\_\_LINE\_\_ // 展開された場所の行番号の数値に置換されます

\_\_func\_\_ // 展開された場所の関数名の文字列に置換されます

\_\_VA\_ARGS\_\_ // 関数マクロで受け取った可変個引数

※python の\_\_init\_\_ とかはここからアイディアを取ってる？ただし，C言語の方は大小文字混じってて秩序がない。

### #undef

定義の取り消し

#undef \_\_HOGE\_\_

### #include

パターンは２つある

#include “hoge.h” //　“” ；自作ファイル（主にヘッダー）を読み込む

#include <stdlib.h> // <>；標準ライブラリを読み込む

### #ifndef

例）

#ifdef

（関連）#ifndef 識別子が定義されていないかどうか = if not def

（参考）<https://www.grapecity.com/developer/support/powernews/column/clang/014/page02.htm>

#error

コンパイル時にエラーメッセージを表示

#line

Cコンパイラに行番号を伝える

### typedef

typedef unsigned int UINT; // UINT と言う新しい型を作った。

typedef unsigned int \_UINT; // アンダーバーで始まる事が多い

// #が付いていない（プリディレクティブではない）点に注意

暗黙の型変換

式内の各オペランドの型が異なる時，一番大きい型に統一される。

型の順位は次のようになる。

char < short < int < long < float < double

## inline　関数のインライン化

インライン展開，コンパイル時に呼び出し元の関数内部に展開される。

実行時の関数呼び出しのオーバーヘッドの減少，メンテナンス性向上などのメリットがある。

（厳密にはコンパイラの制約に従う事になり，必ずしも内部展開されないらしい。）

inline int func(int x) {

return x + (int)(x \* 0.05);

}

int main() {

cout << func(100);

return 0;

}

C言語では#define を使ったマクロでしか対応できなかったが，マクロには問題が多かったのでこのような方法が取られたのでは，と勝手に納得している。20.06.17

## C++特有

### #pragma

機種依存のいろいろな設定を行う命令。

プログラム全体に影響を与えずに，プログラム特定部分でコンパイラの動作を制御する。

#### pragma once

おなじファイルを2回includeしないようにする。

### メモリ関連

malloc メモリの確保

構文：void \*malloc(size\_t size);

例：

#include <stdio.h>

#include <stdlib.h>

char \*str = (char \*)malloc(100); // 100文字まで対応可能な文字列

int \*data = (int \*)malloc(sizeof(int) \* 5); // 要素数5のint配列を作成

= malloc(sizeof(struct Sample)); // 自作クラスや構造体の場合

memory allocation（メモリ割り当て）の意

補足：基本情報技術者試験に「領域割付け関数」という単語がでてくるが、mallocの事などを指していると思われる。

ただし、個人的にはその試験以外では聞いた事がない単語ではある。

free メモリの開放

init 確保したメモリの初期化

### class

後述 （参考）[class](#_class)

## マイコン系特有

### asm アセンブリ言語の使用

構文：asm ( *assembly\_literals* )

通常，短いアセンブリプログラムを記述する。

例）

// standard inline assembly

asm ("movq $60, %rax\n\t"　// the exit syscall number on Linux

"movq $2, %rdi\n\t"　// このプログラムは２を返す。

"syscall");

# プログラム構文

## 変数

複数の変数を同時に宣言する

SomeClass \*hoge, \*fuga; # コンマ(,)で区切る

### extern

通常の関数定義をするスコープで使用

グローバル変数定義を行う？？

## 条件式

## ループ

while

while(1){

}

## データ型関連

sizeof

構文：sizeof(*data\_type*)

# 構造化

## method

引数なしメソッド

void some\_method(void)

{

}

括弧内にvoid を書く必要がある。

## class

特徴：

C言語では使用できない。C++から導入された機能。

通常のオブジェクト指向のクラスと思えばよい。

### 書き方

class person{

// アクセシビリティ

// public 以下は全部公開される

public:

int id;

int age;

std::string name;

// 情報を出力する関数

void

print(){

std::cout << "id:" << id << std::endl;

std::cout << "age:" << age << std::endl;

std::cout << "name:" << name << std::endl;

}

};

person homu = { 0, 14, "homu" }; // 引数付きコンストラクタに相当？

homu.print(); // クラス内メソッドの実行

## struct（構造体）

特徴：

C言語で生まれmethodを持つ事ができなかった為に，メンバ変数のみで構成される印象が強い。

但し，C++ではmethodメンバを持つ事ができる為，class とほとんど変わらない。

違いはデフォルトのアクセシビリティ：classだとprivate structはpublic

⇒ こういった、C++の「制約の広すぎる仕様」はよろしくないと思う。

こんな感じで，各メンバのbyte数？を指定できるらしい 2021-07-27

struct

{

ADS1015\_COMP\_QUE comp\_que :2;

ADS1015\_COMP\_LAT comp\_lat :1;

ADS1015\_COMP\_POL comp\_pol :1;

ADS1015\_COMP\_MODE comp\_mode:1;

ADS1015\_DR dr :3;

ADS1015\_MODE mode :1;

ADS1015\_PGA pga :3;

ADS1015\_MUX mux :3;

ADS1015\_OS os :1;

} reg;

### 匿名構造体（？）

構造体の名前を付けずに，構造体の宣言と同時に変数も宣言する記述。

これは知ってる限り，C++でしか見たことがない。

struct {

int id;

char name[20];

} taro;

※但し非常に読みにくいので避けた方が良いと思う。

## union（共用帯）

構造体に似ているが，共用帯は様々な型のデータを**共通のメモリー領域で管理**する。

例）良くあるのがマイコンでのビット管理

struct char\_bits { // 8bit つかう構造体

char b0:1; // b0:1 で1bit目にアクセスできる？？？

…

char b7:1;

};

union char\_byte {

struct char\_bits　bits; // 8bit の内，１つ１つのビットにアクセスする為のメンバ

char byte; // bit メンバと開始アドレスを共にする(char = 1byte[8bit)

};

# 代表的なエラー

redeclaration of ‘*arg\_name* ' 同じ名前の変数を２回宣言している。

# ライブラリ

## 概要

各標準ライブラリのありか

Linux /usr/include/

## <cstdlib>

abort

構文：void abort (void);

## <iostream>

## <pthread.h>

※コンパイル時に -l pthread　が必要

pthread\_create

スレッドを生成する。int pthread\_create( pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine)(void\*),void \*arg)

thread 作成したスレッドのハンドルを格納するバッファを指定する。

attr スレッド属性を指定する。デフォルト属性でよい場合は NULL を指定する。

start\_routine 新しいスレッドのエントリポイント。

arg 新しいスレッドに渡す引数。必要ない場合は NULL を渡せばよい。

（参考）<http://manpages.ubuntu.com/manpages/trusty/ja/man3/pthread_create.3.html>

pthread\_join

対象スレッドの終了を待つ

構文：

参考

Segmentation fault アクセス禁止メモリへのアクセス

## <stdlib.h>

EXIT\_FAILURE 失敗した事を示す終了コード

malloc

rand

srand

## <stdio.h>

標準入出力を扱う

### printf

構文：int printf ( const char \* format, ... );

注意　数値などを扱いたい時はフォーマット文字を使う

printf("int\_val: %-5d ", int\_val);

|  |  |  |  |
| --- | --- | --- | --- |
| 文字 | 使える型 | 用途 | 使用例 |
| %c | char | １文字を出力する | "%c" |
| %s | char \* | 文字列を出力する | "%4s", "%-8s" |
| %d | int, short | 整数を10進で出力する | "%4d","%-4d","%04d" |
| %u | unsigned int, unsigned short | 符号なし整数を10進で出力する | "%4u","%04u" |
| %o | int, short, unsigned int, unsigned short | 整数を8進で出力する | "%06o","%03o" |
| %x | int, short, unsigned int, unsigned short | 整数を16進で出力する | "%02x", "%4x", %02X |
| %f | float | 実数を出力する | "%8.2f" |
| %e | float | 実数を指数表示で出力する | "%8.3e" |
| %g | float | 実数を最適な形式で出力する | "%g" |
| %ld | long | 倍精度整数を10進で出力する | "%-12ld" |
| %lu | unsigned long | 符号なし倍精度整数を10進で出力する | "%12lu" |
| %lo | long, unsigned long | 倍精度整数を8進で出力する | "%12lo" |
| %lx | long, unsigned long | 倍精度整数を16進で出力する | "%08lx" |
| %lf | double | 倍精度実数を出力する | "%10.3lf" |

### perror

構文：void perror ( const char \* str );

エラーが発生した場所で使い，標準エラーにメッセージを表示する。

## <stdint.h>

uint32\_t

useconds\_t 恐らく

## <string.h>

memset

strlen char配列の長さを返す

## <time.h>

time\_t

例：

time\_t t = time(NULL);

printf("time.time:%ld\n", t);

注意点

time\_tではμ秒などがとれない。μ秒まで取りたければ<sys/time.h>のgettimeofdayを利用する。

（参考資料）<https://www.mm2d.net/main/prog/c/time-01.html>

wait\_nanosecond

### tm構造体

struct tm {

int tm\_sec; /\* 秒 (0-60) \*/

int tm\_min; /\* 分 (0-59) \*/

int tm\_hour; /\* 時間 (0-23) \*/

int tm\_mday; /\* 月内の日付 (1-31) \*/

int tm\_mon; /\* 月 (0-11) \*/

int tm\_year; /\* 年 - 1900 \*/

int tm\_wday; /\* 曜日 (0-6, 日曜 = 0) \*/

int tm\_yday; /\* 年内通算日 (0-365, 1 月 1 日 = 0) \*/

int tm\_isdst; /\* 夏時間 \*/

};

## <winnt.h>

# コンパイラ

## コンパイルの概要

実行ファイルを作る「コンパイル」は一般的に以下の流れで行われる。

プリプロセス

#define， #include，#ifdef　などの処理を行う

gcc -E main.c # プリプロセス後のソースファイルを表示する

コンパイル

プリプロセス済のファイルから，アセンブラソースファイル（.s）を作成する

（補足）アセンブラ：01の羅列である「マシン語」を「ニーモニック」と呼ばれる文字に変換したもの。人間にも可読な状態。

gcc -S main.c # アセンブラソースファイル(.s) に変換

> main.s が作成される

# 蛇足：デバッグオプションを付けると行数（＝ファイルサイズ）が増える

gcc -S main.c -g # 825バイト → 12367バイト に変化

アセンブル

オブジェクトファイル（.o）を作成する。

gcc -c main.c # オブジェクトファイル(.o)を作成

as -o main.o main.s # asアセンブラソース(.s)→オブジェクトファイルを作成。（結果は上と同じ）

> main.o が作成される。

# 蛇足：バイナリ（マシン語）の状態を確認。この状態でも読める人が存在するらしい。

od main.o #

リンク

広義には実行ファイルを作成する事，狭義ではオブジェクトファイルから実行ファイルを作成する事を指す。

gcc main.o # オブジェクトファイルから実行ファイルを作成。

gcc main.c # 通常はソースファイルから一気に実行ファイルを作成する。

# 蛇足：依存ライブラリを見る

ldd ./a.out #

依存ライブラリの補足情報

d-linux-x86-64.so.2 リンクローダーと呼ばれる。実行時に依存ライブラリをメモリ上に持ってきて結合してくれる。

libc.so 定番中の定番の C ライブラリ。

linux-vdso.so Linux 上で動くバイナリに付与される仮想的なライブラリ。高速化のために存在しているらしい。

（参考）<https://aoking.hatenablog.jp/entry/20121109/1352457273>

## gcc

### 概要

GNU Compiler Collection

GNU（完全にフリーソフトで構成されるOS）用のC言語コンパイラ。

近年，直接使う事は少ない。makefileの中で使われる。

### チュートリアル

### コマンド

#### 構文

gcc [options] *source\_file.c*…

gcc main.c hello.c -o main # 複数ファイルをコンパイルする場合

#### オプション（gcc/g++共通）

-o *bin\_name* 指定したファイル名で出力を行う。省略すると a.outというファイルが作成される。

-g デバッグ情報を付加する。dbx,gdbなどのデバッガ を使用するときに必要。--debug

-c オブジェクトファイルの作成のみ。filename.o のみ作成。（実行ファイルは作成されない）

補足：--help文は「リンクは行わない」という表現だったが「リンク」の意味が分からない2021-10-21

-l ライブラリを指定。

gcc main.c -l pthread # pthreadというライブラリ（pthread.h）を利用する

-lm libm.soという共有ライブラリをリンクする。（math.hなどが含まれる）

（補足）-lxxxの場合、リンカはまずlibxxx.soをリンク。libxxx.soが無ければ、libxxx.aというファイルを探してリンク。

-L *linker* ライブラリ探査の対象ディレクトリを追加する

-Wall コンパイル時に警告を出す。（通常はエラーメッセージしか出ない）

-shared 共有ライブラリ（.soファイル）を作成する。

gcc -shared main.c -o libhello.so

以下、未確認情報

-I ヘッダのディレクトリパスを指定

-L ライブラリのディレクトリパスを指定

### フラグ：$()

#### 概要

基本的にmakeで用い，コンパイル時にフラグを指定する？

gccの場合は $(CFLAGS) となり，g++の場合は $(CXXFLAGS) となる（らしい：2021-10-21）

2021-10-08 $‘’でも良いらしい？

#### 補足

gccの場合

main.c から $(CC) -c $(CPPFLAGS) $(CFLAGS) という形式のコマンドで自動的にmain.o → a.outを作成

g++の場合

main.cpp から $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) という形式のコマンドで自動的にmain.o → a.outを作成

CPPFLAGSはC PreProcessorの意であって，c++の事ではない。要するにプリプロセッサ用のフラグ。

一方，CFLAGSとCXXFLAGSはコンパイラ用のフラグ。

## g++

### 概要

GNU Compiler Collection

GNU（完全にフリーソフトで構成されるOS）用のc++コンパイラ。

### チュートリアル

基本的にはgccと同じ。

[ main.cpp ]

#include <stdio.h>

int main(int arg, char\* argv[]){

printf("hello\n");

}

terminal

g++ main.cpp -o hello

### コマンド

#### 構文

g++ [オプション] *source\_file.cpp*…

#### オプション（g++特有）

-std C++11, C++14 など新しい C++規格の機能を利用する。

### トラブルシューティング

#### fatal error: Killed signal terminated program cc1plus

原因：メモリ，スワップ領域の不足。またはコンパイルに使うコア数が多すぎる。

対応

・メモリかスワップ領域を増やす。

・コア数を減らす

make -j1

（情報元[英語]）<https://github.com/mlpack/mlpack/issues/2775>

## ldd

ldd a.out

a.out:

lib.so.7=> /lib/libc.so.7 (0x80081b000)

## makefile

### 概要

c/c++のコンパイルを楽にする為の仕組み。小～中規模のプロジェクトで用いられる事が多い。

（補足）大規模ではcmake，極小規模ではgccやg++を直接使う事が多い。

補足

厳密にはmake install でインストールできる訳では無く、「install」というターゲットを実行しているに過ぎない。

関連資料

configure,make, make install について：<https://qiita.com/chihiro/items/f270744d7e09c58a50a5>

### チュートリアル

ファイル構成

├ main.c

├ hello.c

├ hello.h

├ makefile

[makefile]

NAME = hello コンパイルで作成されるファイル名

CC = gcc C言語に使用するコンパイラ

all: 「ターゲット」を指定しなかった場合の処理

$(CC) main.c -o $(NAME) 　実際の処理を記述。（※１）

test: main.c hello.c 「ターゲット」名が「test」の場合

$(CC) main.c hello.c -o $(NAME)

補足

※１　ここの$(CC) などは上で指定したマクロ（≒変数）で，結果としてこの行は　 gcc main.c -o hello となる。

terminal

make # ターゲットとしてallが選択される

> gcc main.c -o hello

make test # 明示的にターゲット名testを指定

> gcc main.c hello.c -o hello

（参考）<https://qiita.com/yagiyuki/items/ff343d381d9477e89f3b>

### configureファイル

厳密には

./configure

で、makeによるコンパイルを想定されているソースコード群と同じディレクトリに用意されているスクリプト。

インストールに必要な環境変数やライブラリが正しく設定、設置されているか等を確認した結果、MakeFileを作成する。

### make

#### 概要

インストール

sudo apt install build-essential # Debian系

#### コマンド

構文：make サブコマンド [オプション] [ターゲット]

サブコマンド

install アプリをインストールする。開発時には使わない。

補足：厳密にはMakeFileに記述された「install」というラベルから処理を実行する　と言うのが正しい。

altinstall

補足：make install は 既存のバイナリを上書きまたはリンクを破壊する可能性があり、開発版やマイナーバージョンのインストールにはmake altinstall が推奨されている。

オプション

-I ディレクトリ インクルードするディレクトリを指定 --include-dir=ディレクトリ

-j[Num] コンパイル時に使用するジョブの数を指定。CPUを並列で使える為，処理が早くなる。

Num省略で制限なし。 例 -j4 --jobs[=Num]

-D

### makefile

拡張子無しで「makefile」と言うファイルを作成する。（ファイル名はMakefileでもいいらしい？）

スクリプト言語みたいになっていて，上半分は変数の定義。⇒ 20.10.19 恐らく，完全にbash

#### ターゲット

（ターゲット）：（依存ファイル）をいう風に書く

[ makefile ]

test:main.cpp

g++ -o test main.cpp

terminal

make test # 実行時には make ターゲット名 とする

|  |  |  |
| --- | --- | --- |
| 項目 | 説明 | 具体例 |
| .c.o: | オブジェクトファイル群の指定？ | $(CC) -c $(CFLAGS) $< -o $@ |
| all: | ターゲット名を指定しなかった場合 | all: test  $(TARGET) |
| test: | よく使われるターゲット名の１つ |  |
| $(TARGET): | TARGETというマクロ（後述）に値を入れた場合 | $(OBJECTS)  $(CC) $(OBJECTS) $(LDFLAGS) -o $@ |
| clean: | make clean としたときに実行されるスクリプトを記述 | -rm -f $(OBJECTS) $(TARGET)  -rm -f \*~ core |

#### マクロ

makefile内で使える変数のようなもの。

そのため，マクロ名は別に何でもよいが慣例的に使われているものがあり，素直にそれを用いた方が他のメンバーとの意思疎通がしやすい。

|  |  |  |
| --- | --- | --- |
| マクロ名 | 説明 | 値の具体例 |
| CC | Cコンパイラ | gcc |
| CXX | C++コンパイラ | g++ |
| CFLAGS | コンパイルの際のオプション | `pkg-config --cflags gtk+-3.0` |
| LDFLAGS | リンク用のディレクトリ | `pkg-config --libs gtk+-3.0` -lm |
| SOURCES | コンパイル対象のソース群 | main.c |
| OBJECTS | リンクさせるオブジェクトファイル群 | $(SOURCES:.c=.o) |
| TARGET | コンパイルで作成されるファイル名 | my\_app |
| INC | インクルードファイルのパス指定 |  |
| LIB | ライブラリファイルのパス指定 |  |

$@

#### 記述例

CC = g++ # コンパイラ

CFLAGS = # コンパイルオプション

TARGET = Sample # 実行ファイル名

SRCS = Sample.cpp # コンパイル対象のソースコード（複数指定可能）

OBJS = $(SRCS:.cpp=.o) # オブジェクトファイル名

INCDIR = -I../inc # インクルードファイルのあるディレクトリパス

LIBDIR = # ライブラリファイルのあるディレクトリパス

LIBS = # 追加するライブラリファイル

# ターゲットファイル生成

$(TARGET): $(OBJS)

$(CC) -o $@ $^ $(LIBDIR) $(LIBS)

# オブジェクトファイル生成

$(OBJS): $(SRCS)

$(CC) $(CFLAGS) $(INCDIR) -c $(SRCS)

# 実行ターゲット

all: clean $(OBJS) $(TARGET)

clean:

-rm -f $(OBJS) $(TARGET) \*.d

### makelist

※ファイル名は拡張子無し

## cmake

### 概要

特にLinuxにおいて，自動でmakefileなどを作成しコンパイルを簡易化する為のツール。大規模プロジェクトで用いられる。

CMakeLists.txt にコンパイルの内容を記述し、bulidディレクトリからcmake .. を実行し，作成されたMakefileを実行(make) する，という流れ。

インストール

sudo apt install cmake

in-sourceビルド

cmake . のようなやり方。非常に簡単。

元のソースコードを格納しているディレクトリにコンパイル後のファイルまで格納されてしまうため管理がめんどくさくなる。

out-of-sourceビルド

### コマンド

構文：cmake [options] [-S <path-to-source>] [-B <path-to-build>]

オプション

-G generator 名を明言する

-DCMAKE\_BUILD\_TYPE=Debug デバッグビルドする場合のcmakeオプション

### チュートリアル

ファイル構成（概要）

├ my\_project .cファイルなど，実際のコードを格納する

├ build コンパイルされたファイルを格納する為のディレクトリ

│ ├ my\_project cmake で作成されるディレクトリ

│ └ Makefile 最終的に実行する事になる、cmakeで自動作成されたファイル

├ cmake cmakeのツール群？

├ CMakeLists.txt コンパイル内容が記述される

├ hoge.cmake CMakeLists内でincludeされるファイル

・対象をディレクトリにすると、そのディレクトリ内の全てをコンパイルする（らしい）

cd build #　ディレクトリ移動。ここでcmakeを実行する事で，コンパイル済のファイルがここに格納される。

cmake .. # 1階層上にあるCMakeListを基にビルド

### CMakeLists

cmake\_minimum\_required(VERSION 3.12) #　バージョンを指定

include(pico\_sdk\_import.cmake) #

project(pico\_examples C CXX ASM) # プロジェクト名を指定

#　CXX はC++の意

set(CMAKE\_C\_STANDARD 11) #　環境変数の指定

set(CMAKE\_CXX\_STANDARD 17)

configure\_file(TutorialConfig.h.in TutorialConfig.h) #　ヘッダ―などの指定？

pico\_sdk\_init() # SDKを初期化

include(example\_auto\_set\_url.cmake)

# Hardware-specific examples in subdirectories:

add\_subdirectory(blink)

（チュートリアル；英語）<https://cmake.org/cmake/help/latest/guide/tutorial/index.html>

#### 環境変数

|  |  |
| --- | --- |
| 変数名 | 意味 |
| CMAKE\_SOURCE\_DIR | ソースコードのトップディレクトリ。cmake ../とかで指定された場所。 |
| CMAKE\_BINARY\_DIR | cmakeの成果物が置かれるトップディレクトリ |
| PROJECT\_SOURCE\_DIR | projectオプションで指定されたプロジェクトのトップディレクトリ。 |
| PROJECT\_BINARY\_DIR |  |
| CMAKE\_INSTALL\_PREFIX | インストール先のprefix。デフォルトは/usr/local。  cmake -DCMAKE\_INSTALL\_PREFIX=pwd/../localといったオプション指定で変更可能 |
| CMAKE\_C\_FLAGS | コンパイラ―に渡したいオプションを指定する。  CMAKE\_CXX\_FLAGS も存在する。 |
| CMAKE\_BUILD\_TYPE | ビルドタイプを指定  値：Debug, |
| ARCHIVE\_OUTPUT\_DIRECTORY |  |
| LIBRARY\_OUTPUT\_DIRECTORY |  |
| RUNTIME\_OUTPUT\_DIRECTORY | runtimeファイルがビルトされるディレクトリを指定 |

こんな指定方法があるらしい

set\_target\_properties( targets...

PROPERTIES

ARCHIVE\_OUTPUT\_DIRECTORY "${CMAKE\_BINARY\_DIR}/lib"

LIBRARY\_OUTPUT\_DIRECTORY "${CMAKE\_BINARY\_DIR}/lib"

RUNTIME\_OUTPUT\_DIRECTORY "${CMAKE\_BINARY\_DIR}/bin"

)

#### コマンド詳細

add\_custom\_target カスタムターゲットの実行

add\_executable 実効ファイルの生成

add\_executable(binary.exe main.cpp sorce1.cpp)

add\_library ライブラリファイルの生成

add\_subdirectory 下層ディレクトリを追加。別々のビルド対象を格納

cmake\_minimum\_required バージョンの指定

configure\_file テンプレートファイルを利用。

target\_link\_libraries リンクするライブラリーの指定

### 資料

推奨記法について：<https://qiita.com/shohirose/items/5b406f060cd5557814e9>

ライブラリの資料①：<https://cplusplus.com/reference/cstdio/perror/>

ライブラリの資料②：<https://code-reference.com/c/stdio.h/perror>

## 資料（コンパイラ関連）

### 暗黙のルール

（本家資料；英語）<https://www.gnu.org/software/make/manual/html_node/Implicit-Variables.html>

### 用語

GCJ GNU Javaコンパイラ

GNAT GNU Adaコンパイラ

### 拡張子

.c （小文字）C言語。プリプロセッサ、コンパイラ、アセンブラにかけられる。

.cxx C++。プリプロセッサ、コンパイラ、アセンブラにかけられる。.C（大文字），.ccでも良い。

.m Objective-C 言語ソース。プリプロセッサ、コンパイラ、アセンブラにかけられる。

.i プリプロセッサにかけられたC言語ソース。コンパイラ、アセンブラにかけられる。

.ii プリプロセッサにかけられたC++言語ソース。コンパイラ、アセンブラにかけられる。

.s アセンブリ言語ソース。アセンブラにかけられる。

.S アセンブリ言語ソース。プリプロセッサ、アセンブラにかけられる。

.h プリプロセッサファイル。通常はコマンドラインには現れない。

.o オブジェクトファイル。≒dll。大規模ソフトなどはオブジェクトファイルごとにコンパイルするのが効率的。

.so 共有ライブラリ