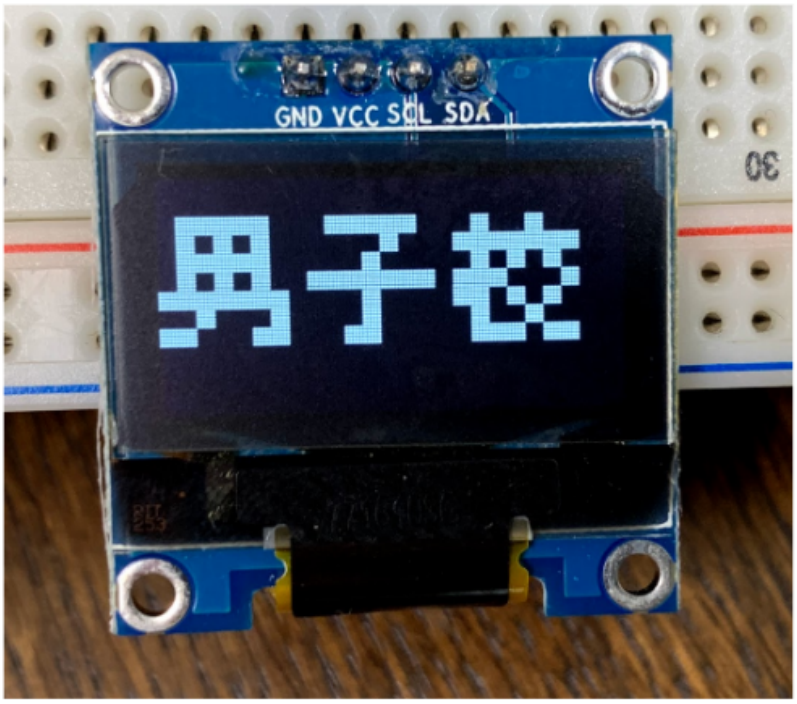
Arduinoテキスト
中級編



目次

1.制御文(発展)

switch case

switch case文はif文同様場合分けに使う。特定の変数に入った値によって場合分けを行うことが出来、うまく使うとif elseを使うよりシンプルに書ける。さいころの出た目によってコメントを変える処理などに使える。

<プログラム例>

```
switch (var) {
  case 1:
    // varが1のとき実行される
    break;
  case 2:
    // varが2のとき実行される
    break;
```

```
  default:
    // どのcaseにも一致しなかったとき実行される
    // (defaultは省略可能)
}
```
※このテキストのサンプルプログラムでは、setup関数やloop関数は省略する事がある。

defaultは一致するものが無いと実行される。また、break;がないと、その下のcaseの中も続けて実行される。

do while

do whileは、whileと似ているが、ループの最後に条件式のテストが行われる。故に、条件式がfalseでも必ず一度は実行される。

<プログラム例>

```
do {
  delay(50); // センサが安定するまで停止
  x = readSensors();
} while (x < 100);
```

2、#include

ライブラリをスケッチに取り入れたい時に使い、「#include <[拡張子が付いたライブラリ名]>」と書く。セミコロンは不要。これにより、Arduino専用のライブラリや、ArduinoのチップであるAVRマイコン用のC言語で書かれたライブラリを使用できる。

<プログラム例>

```
#include <Wire.h>
```

3、?:（条件演算子）

条件式？真の時：偽の時

と書きます。

例

```
return (x==y)?1:0
```
これはx==yなら1をそれ以外なら0を返します。

flagがtrueならaに、falseならbにするとかなら

```
char x = flag?'a':'b';
```
ってなります。

~~(見づらい...)~~

4 データ型（発展）

char

文字列リテラル

char型を理解するために、「文字列リテラル」を説明する。

char型は1バイト(つまり256文字分)を消費して英数字1文字を記憶する。しかし、1文字を記憶するといってもコンピュータは0と1しか扱えないので、文字を代わりの数字で表す必要がある。だから、「Aは65」というように、文字を数字に変換して処理がなされている。

だから、Aをchar型の変数で記録し表示するには、

```
char myChar = 65;
Serial.print(myChar);  // Aと表示される
```

とする方法がある。

ところが、プログラム上で数字の羅列でないと文字を入力出来なかったら億劫だ。そこで登場するのが「文字列リテラル」である。

文字列リテラルというのは、' '(シングルクォーテーション)や" "(ダブルクォーテーション)で囲むことで記述する文字列であることを表す定数で、数字の代わりに直接文字を入力できる。(「リテラル」とは「文字通り」という意味がある。)だから、

```
char myChar = 'A';
Serial.print(myChar);  // Aと表示される
```

と記述出来る。

ASCIIコード

前項で「文字を数字に変換して処理がなされている」と述べたが、半角の英字（a～z、A～Z）やアラビア数字（0～9）、記号、空白文字、制御文字など128文字が規定されている「ASCIIコード」が使用されている。本来、char型は256種類記録出来るが、最上位は符号ビットとされており、実際には(0~+127)の値に文字が規定されている。そのため、"A"+1を表示させようとすると、66番のBが表示される。

```
char myChar ='A'+1;
Serial.print(myChar);  // Bと表示される
```

ただし、同じく8ビットのbyte型は、Serial.printすると、保存されている数字がそのまま出力される。これは、基本型の変数はメモリ上で変数の型と変数名、値がセットで保存されており、Serial.printでは、char型等の文字用の変数でないと文字に値を変換して出力しないからだ。

```
byte myChar = 65;
Serial.print(myChar);  // 65と表示される
```

unsigned char

char型は符号ビットがある型で、-128~127の値のうち、0~127をASCIIコードとしている。それは、最上位の符号ビットは0のままで使用する事を示す。

それに対し、unsigned charは符号ビットがないため、0~255の値を扱う。また、char型と異なり、文字を保存ことが出来ない為、bool型とまったく同じである。

```
void setup() {
  char myChar1 = 'A';
```

```
  unsigned char myChar2 = 'A';
  Serial.begin(9600);
  Serial.println(myChar1);  // Aと表示される
  Serial.println(myChar2);  // 65と表示される
}

void loop() {}
```

故に、一貫性を保つためにbool型を使うべきだ。

unsigned int（符号なし整数型）

unsigned int型は、2バイトの値を格納する点ではint型と同じですが、負の数が扱えず、0から65535までの正の数だけを格納します。

符号付き整数型と符号なし整数型の違いは、最上位ビットの解釈の違いです。

<プログラム例>

```
unsigned int ledPin = 13;
```

long（long整数型）

long型の変数は32ビット(4バイト)に拡張されており、-2,147,483,648から2,147,483,647までの数値を格納できる。

<プログラム例>

LEDがゆっくり点滅する。周期はおよそ5秒である(クロックが16MHzのArduinoの場合)

```
long counter = 0;

boolean led = false;

void setup() {
  pinMode(13, OUTPUT);      // ピン13のLEDを点滅
  digitalWrite(13, led);
}
void loop() {
  if(++counter == 1000000) { // 100万回に1回
    led = !led;              // LEDを反転
    digitalWrite(13, led);
```

```
    counter = 0;
  }
}
```

unsigned long（符号なしlong整数型）

32ビット（4バイト）の値を格納する変数。つまり、(2の32乗 -1)、0～4294967296の値を格納できると言う事。
主には、millis()の値を格納するのに使用する。使用例は、mllis()の項目を参照すると良い。

float（浮動小数点型）

浮動小数点: 「2.236」を「2236×10^-8」と表す様に、仮数(2236)、指数(-8)、基数(正か負か)で小数を表す事。
float型の場合、4バイト、つまり32ビットを使いそれらを表す。
←上位ビット(合計32ビット)

符号	指数を8ビットの2進数で表す	過数を23ビットの2進数で表す

また、浮動小数点を使った演算を行うと誤差が生まれるので注意が必要だ。
例えば、0.01を浮動小数点で表すとする。するとコンピュータはまず、0.01を2進数で表そうとする。すると、0.01(10)=0.00000010100011110101110000101000…(2)
という循環小数になってしまう。意外に感じられる方もいらっしゃるかも知れないが、ちゃんと計算をするとこうなる。
すると、下位のビットの数は捨てられ、実際の0.01より小さい数になるため、float型の0.01を10回足しても0.1以下になってしまう。
<プログラム例>

```
float x = 0.01;
int y = x * 100;  //y<1となる。( float型の変数をint型に代入すると小数点以下は切り捨てられる。)
```

<プログラム例>

```
int x, y;
float z;
x = 1;
y = x / 2;          // yは0(整数型÷整数型＝整数型になり、整数型は小数点以下をきりすてる)
z = (float)x / 2.0; // zは0.5(2ではなく、2.0で割っている)
```

double（倍精度浮動小数点型）

Arduino Unoのdouble型はfloat型と同一の実装で、この型を使っても精度は向上しない。double型を含むコードをArduinoへ移植する際は注意する。
Arduino Dueにおけるdouble型は8バイト(64 bit)の精度である。

文字列（配列）

文字列は2つの表し方が有り、1つはchar型の配列とヌル終端というもので終端を表す組み合わせを使う方法、もう1つはStringクラスを使った物で、連結、追加、置換、検索等を行う事が出来るが前者よりメモリを消費する。
ここでは、従来型である前者を説明する。基本的な使い方は、一般的な配列と変わりない。
例：

```
char Str1[15];  // 15文字分記録する配列の宣言。
char Str2[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o'};  // 「文字列リテラル」を使い、通常の配列と同様に文字を入れる。
char Str3[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o', '\0'};  // ヌル文字も入力した。
char Str4[ ] = "arduino";  // ダブルクォーテーションで囲むと連続して文字を入力でき、配列として代入出来る。
char Str5[8] = "arduino";
char Str6[15] = "arduino";  // 余白を残して初期化
```

配列の要素の数が実際に入力したいArduinoの文字数(7文字)よりも多く設定されている事にお気づきだろうか。これは、oの文字の配列の後ろの要素にヌル文字(ASCIIコードの0で、表記は「\0」)が入っているためである。ヌル文字は文字列の終端を表すアスキー文字の制御文字の1つである。これがないと文字列以降のメモリも読み込んでしまうという問題が発生する。ちなみに1文字(文字列リテラルで' 'の場合はヌル文字はつかない。宣言する際に必要以上の要素数を確保するのは問題ないが、ヌル文字を含めない要素数での宣言をしてはいけない。尚、要素数を指定しなくても自動的に丁度良い数が入る。" "も文字列リテラルの1つで、char型の配列をソースコード上に直接入力するための技法だ。

5、ライブラリの保存場所（基礎）

標準ライブラリ

標準ライブラリは、Arduino IDEに元々含まれていてIDEのアップグレードと同時にアップグレードされる場合がある。
例)EEPROM、SoftwareSerial、Stepper、Wire、SPI、Servo、LiquidCrystal、SD Library

保存場所は、
IDEインストールフォルダの”libraries”サブフォルダ
/Applications/Arduino.app/Contents/Java/libraries/(Mac)
\Program Files\WindowsApps\ArduinoLLC.ArduinoIDE_1.8.33.0_x86__mdqgnx93n4wtt\libraries(Windows)

非標準ライブラリ

ZIP形式等でダウンロードしたライブラリ。例)エイダフルーツのライブラリ
保存場所は、

スケッチブックフォルダ内の”libraries”サブフォルダ
C:\Users\takeh\OneDrive\ドキュメント\Arduino\libraries(Windows)
/Users/{username}/Documents/Arduino/libraries/(Mac)

6　変数の応用

Fマクロ

Flashメモリに文字列を記録する。通常、Arduino上の変数等を記録するSRAM(メモリ)に文字列等のデータは記録されるが、それは容量が小さい為、本来プログラム等を保存するFlashメモリに記録する。無くても動作する。
ディスプレイに大量の文字を表示する際などに有効。

```
void setup() {
    Serial.begin(9600);
}
void loop() {
    Serial.print( F("Hello World.") );
    delay(1000);
}
```

Cast

castは変数の型を別の型に変換したいときに使う。使い方は「(変換後の型名)変換したい変数名」である。

```
int i;
float f;
f = 3.6;
i = (int) f; // この場合、3がiに代入される。
```

int型とint型の割り算は整数になってしまうが、
double型とint型の割り算は小数になる。

そのため

```
int x=13;
int y=8;
```

x/y　は1になるが。。。
(double)x/y　は1.625になる。
xをdouble型にしてそれをint型のyで割っているからです。

次の場合はどうでしょう
(double)(x/y)　は１になる

x/yの結果をdouble型にしていて、
１をdouble型にしているから。

問題

int型の変数a,bを作って8÷3の答えをfloatでシリアルモニターに表示するプログラムを書いてください。答えを表す関数の名前は何でも構いません。

答え

例

```
int a = 8;
int b = 3;
int ans;
void setup(){
      Serial.begin(9600);
  }
void loop(){
      ans = (float) a/b;
      Serial.println(ans);
  }
```

7 時間に関する関数（発展）

millis()

プログラムを開始して何ミリ秒経ったかを返す。
戻り値、つまりmillis()に入ってる数が、開始してからの時間である。(型はunsigned long)。約50日でオーバーフローする＝0に戻る。

<経った時間を表示するプログラム例>

```
unsigned long time;

void setup() {
  Serial.begin(9600);
}

void loop() {
  time = millis();
  Serial.println(time);

  delay(1000); // 1秒おきに経った時間を送信
}
```

<1分間に何回「K」と送られてきたかをしめすプログラム>

```
int point;
void setup() {
  Serial.begin(9600);
  while(millis() < 60000) {
    if ('K'==Serial.read()) {
      point++;
    }
  }
  Serial.println(point);
}

void loop() {}
```

問題

1. 経った時間を5秒毎に表示するプログラムを作って下さい。（シリアルモニターに表示する）
2. 1秒毎にLEDが段々明るくなるプログラムを作って下さい。
   （analogWriteを利用し、A3ピンを使う事とします）（明るくなる際の数値は自由です）

答え

1. 例

```
unsigned long time;

void setup(){
	Serial.begin(9600);
}

void loop(){
	time = millis();
	Serial.println(time);
	delay(5000);
}
```

2. 例

```
unsigned long time;

void setup(){
	Serial.begin(9600);
}

void loop(){
	time = millis();
	analogWrite(3, time/20);
	delay(1000);
}
```

micros()

分解能となります。

1,000マイクロ秒は1ミリ秒、1,000,000マイクロ秒は1秒です。

<プログラム例>

起動からの時間(マイクロ秒)をシリアルで送信します。

```
unsigned long time;

void setup() {
```

```
  Serial.begin(9600);
}
void loop() {
  time = micros();
  Serial.println(time);
  delay(1000); // 1秒おきに送信
}
```

問題

delayMicroseconds(us)

プログラムを指定した時間だけ一時停止します。単位はマイクロ秒です。数千マイクロ秒を超える場合はdelay関数を使ってください。

現在の仕様では、16383マイクロ秒以内の値を指定したとき、正確に動作します。この仕様は将来のリリースで変更されるはずです。

<プログラム例>

1周期が100マイクロ秒のパルスでLEDを点灯させます。

```
int outPin = 13;                 // LEDはピン13に接続
void setup() {
  pinMode(outPin, OUTPUT);       // 出力として使用
}
void loop() {
  digitalWrite(outPin, HIGH);    // LEDを点灯
  delayMicroseconds(50);         // 50us停止
  digitalWrite(outPin, LOW);     // LEDをオフ
  delayMicroseconds(50);         // もういちど50us待つ
}
```

補足

この関数は3マイクロ秒以上のレンジではとても正確に動作します。それより短い時間での正確さは保証されません。

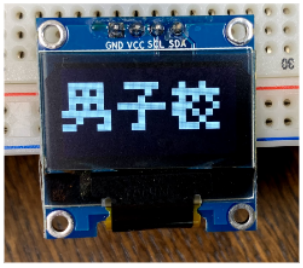
Arduino0018から、この関数は割り込みを停止しない実装になりました。

問題

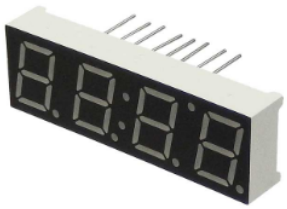
8 OLEDディスプレイ

電工研で使うOLEDディスプレイ

電源電圧: 3~5.5V
制御方式: I2C
制御チップ: SSD1306
解像度: 128×64
表示色: 白
I2Cアドレス: 0x3C



OLEDディスプレイか、7セグメントLEDか



OLEDディスプレイの様な表示器として、7セグメントLED(右図)がある。これは、7つのセグメント(光る部分)の明暗ので1から9までの数字を表す。それらを並べる事で大きな数字もあらわすこともできる。LEDではないが、電卓の液晶表示も同様だ。一見して、128×64ものドットを制御するより、7セグメントLEDの方が単純に見えるだろう。
ところが、電工研では実際の所、単純な数値表示さえもOLEDディスプレイを使うことが多い。それは何故だろうか。その答えは「7セグメントLEDは制御が面倒でピン数を食うから」である。7セグメントLEDはダイナミック制御という配線・制御を行う事でピン数を減らすことができたり、専用のICと組み合わせたり、シリアル通信をしようする物を使ったりして使うのが一般的だが、いずれの場合も制御が煩雑だったり、使うピン数が多かったりする。それに対し、このOLEDディスプレイはI2C通信、専用のライブラリを使うことで、信号線は2本のみで簡単に制御出来る。

通信

SSD1306(制御チップ)をI2Cという規格のシリアル通信を介して制御する。(詳しくは『シリアル通信』の項目を参照。)I2Cなので、マスタ(Arduino)がスレーブ(ディスプレイ)とタイミングを合わせるためのクロック信号を送るSCLピン、情報を相互に送るSDAピン、GNDピン、VCCピン(電源)の計4ピンを使用する。ちなみに使用にある「I2Cアドレス: 0x3C」というのは、通信相手を識別するために、個々のマスタが持っているアドレスのことで、通常は送信したいデータの直前に送信する。つまり、宛名。また、I2CやSPIなどのシリアル通信全般の注意点として、離れた距

離で通信するとノイズが入り乱れる事がある。尚、電線はより線より単線のほうがノイズは出にくく、電線にアルミホイルを巻くと軽減される。
接続
ディスプレイ　Arduino
SCL----------SCL
SDA----------SDA
VCC----------5V
GND----------GND
※ArduinoのSCL、SDAピンはリセットボタンの近く。
※ESPは#4がSDA、#5がSCL。

ライブラリ

このOLEDディスプレイを簡単に使うために、Wire、Adafruit_GFX、Adafruit_SSD1306の3つのライブラリ利用させて頂く。
Wire.hは、I2Cを簡単に行う為のライブラリ。
Adafruit_GFXは、多くのグラフィック表示器に使うことができ、点や直線、円等の表示を関数に座標などの情報を引数を入れるだけで簡単に行う事ができるライブラリ。
Adafruit_SSD1306は、このOLEDの制御チップであるSSD1306を制御するためのライブラリ。
ちなみに、Adafruitというのは、モジュール等を製造・販売しているメーカーで、「エイダフルーツ」と読む。お世話にになります。

文字の表示

<プログラム>

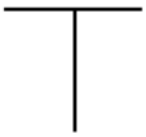
```
#include<Wire.h>
#include<Adafruit_GFX.h>
#include<Adafruit_SSD1306.h>

// インスタンスの生成
Adafruit_SSD1306 display(-1);

void setup() {
  // ディスプレイの初期化
  display.begin(SSD1306_SWITCHCAPVCC, 0x3C);
}

void loop() {
  // ディスプレイをクリア
  display.clearDisplay();

  // 出力する文字の大きさ。テキストサイズ1で、縦のドット数が32ドット。
  display.setTextSize(2);
```

```
  // 出力する文字の色
  display.setTextColor(WHITE);
  // 文字の一番端の位置。左上のマスが原点。x軸は右方向、y軸は下方向が正の向き。
  display.setCursor(0, 0);

  // 出力する文字列
  display.println("Hello");
  display.println("World!");

  // ディスプレイへの表示
  display.display();
  // 1000msec待つ
  delay(1000);
}
```
</プログラム>

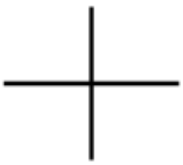
関数の呼び出し部分については、プログラムのコメントアウトを見ればわかると思う。
display.~はAdafruit_SSD1306やAdafruit_GFXにある関数を呼び出しており、呼び出しの時に使う「display」の部分は「インスタンスの生成」で指定している。
※ライブラリはC++で書かれており、.h(ヘッダファイル)と、.cpp(ソースファイル)で構成される。ヘッダファイルは目次のようなもので、ソースファイルにその内容が書かれている。ヘッダファイルがないとソースファイルに関数を書いてもそれは存在しない扱いになる。

9、回路図

回路図とは



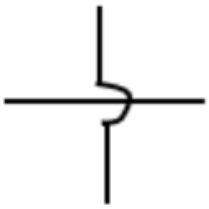
電気回路(回路)=導線で電源や　　ぎ、電気が流れる経路を作った物



回路図＝電気用図記号を用いて回路を表した図

回路図が表す内容

→部品の種別
→部品の定数
→部品同士の接続



回路図の書き方

接続と交差

接続されている	接続されていない
T字交差	黒丸無し十字交差
黒丸付き十字交差)紛らわしいのでT字交差をできるだけT字交差を用いる。)	いずれかの線が孤を描いている十字交差

・電圧が高い方を上側に、低い方を下側に書く。
　→電気は電圧が高い所から低い所へ流れるので、電気の流れが理解し易くなる。
　ただし、これ以降のルールとの兼ね合いもあるので、絶対的なもので　はなく、見やすくなるように調整する。
・左から、入力→処理→出力という流れにする。
　→フィードバック制御の様に、例外的に入力側に戻る配線を書く際　は、他の線と被らないようにする。
　→例）赤外線センサー→Arduino→モータードライバ→モーター
・機能ごとにまとめる。
　→例えば、入力の中でも、センサー、電源、書き込み端子等に分ける。