講師用

ロボット博士養成講座

ロボティクスプロフェッサーコース

不思議アイテムⅢ－２②

（第3回/第4回テキスト）

必ず、生徒に授業日と自分の名前を記入させるようご指導をお願いいたします。

第３回授業日 2024年　　月　　日

第４回授業日 2024年　　月　　日

名　前

Human ヒューマンアカデミー ジュニア STE"AMスクール

ロボット博士養成講座 ロボティクスプロフェッサーコース

2024年11月授業分

ロボット博士養成講座

ロボティクスプロフェッサーコース

不思議アイテムⅢ-2②

第3回

メモリーとビットと演算

講師用

目　　次

0.　メモリーとビット演算

　0.0.　「メモリーとビット演算」でやること

　0.1.　必要なもの

1.　デジタルの世界について

　1.0.　コンピューターの仕組み

　1.1.　デジタルデータとは

　1.2.　画像のデジタルデータ

　1.3.　16 進数

2.　液晶ディスプレイにキャラクターをかく

　2.0.　データ型の変数

　2.1.　白黒画像をかく

　2.2.　カラー画像をかく

3.　ビット演算

　3.0.　ビット演算とは

　3.1.　ビット演算の種類

4.　まとめ

○　授業開始にあたって

　授業のはじめは、着席させ、大きな声であいさつしてから始めます。

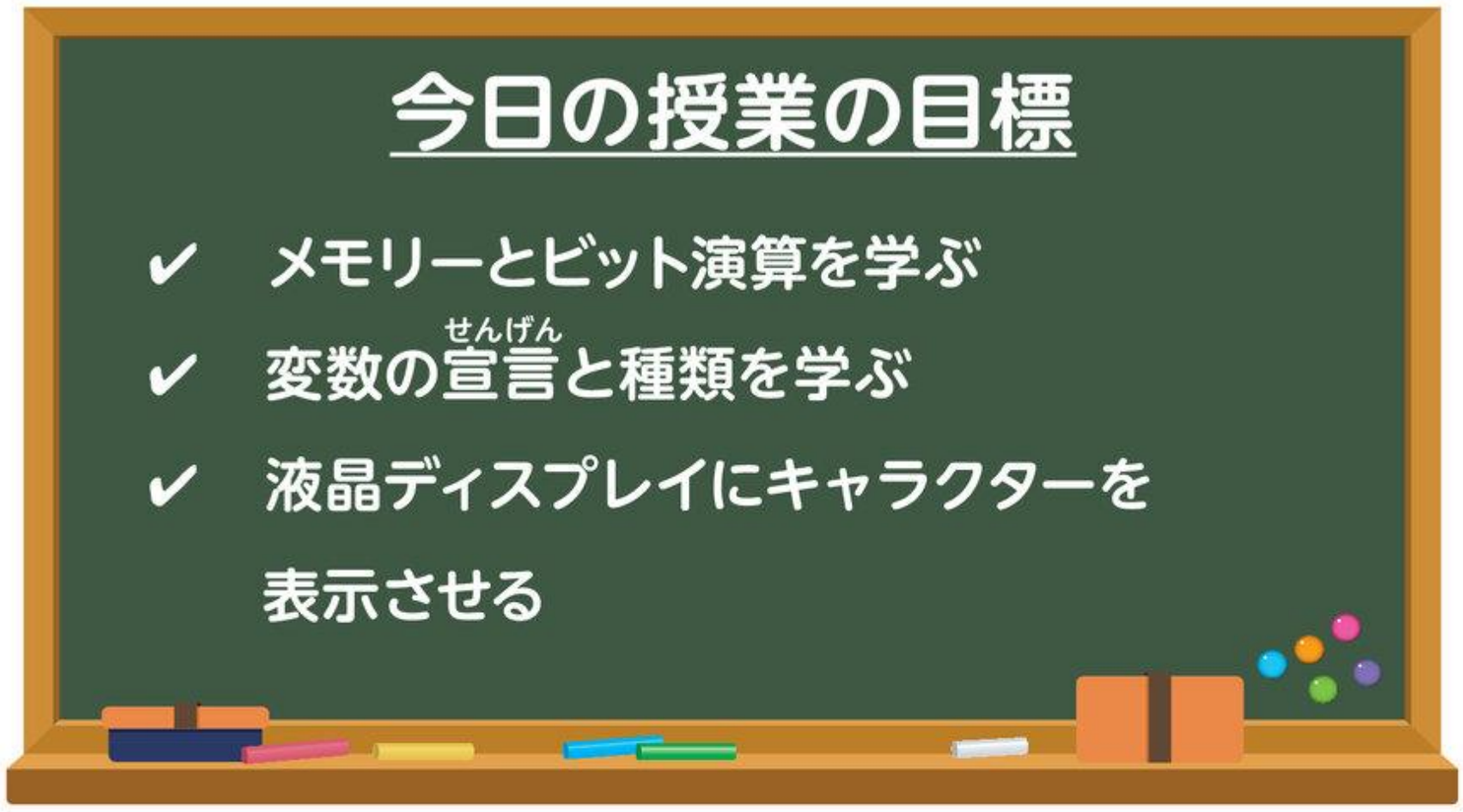
○　今回の目標をパネルで用意するか、黒板に予め書いておきます。

　（授業の目標を明確化することは大変重要なことですので、生徒によく理解させます）

目安時間は授業時間 120 分のうち、休憩 10 分程度取ることを想定しています。
生徒の進捗状況により、休憩時間などを調整して授業を行ってください。

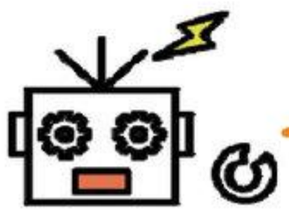
0.　メモリーとビット演算（目安５分）

0.0.　「メモリーとビット演算」でやること



今回は、摩訶不思議なコンピューターの世界について勉強しましょう。
皆さんがパソコンやスマートホンを使うときに、「メモリー」という言葉を聞いたことはありますよね。英語でいうメモリー（memory）は記憶とか思い出という意味ですが、コンピューター機器のメモリーは一般的に「主記憶装置」を意味します。では、どんなことを記憶しておくのかというと、前回の授業でやってみたスイッチが押されたといった状況などです。
今回は、このメモリーのしくみを中心に勉強しましょう。いつもとちがって少しお勉強めいた内容にはなりますが、この知識が液晶ディスプレイを使ったゲームづくり、そしてプログラミングを通じたコンピューターへの理解に大きく役立ちます。がんばりましょう！

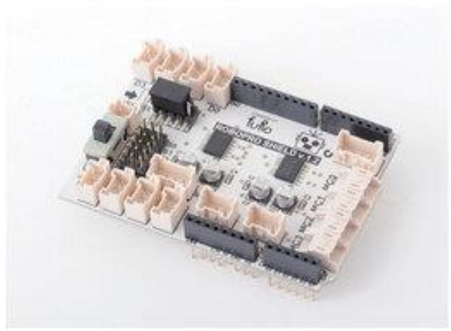
図0-0　コンピューター機器のメモリー



マイコンくんの脳みそをちょっとのぞいてみよう。

0.1.　必要なもの

以下が今回使用するパーツです。
液晶ディスプレイは、パソコンのUSBからの給電だけでも動作しますが、電池ボックスを使う場合は**図0-2**のようにロボプロシールドを使いましょう。

USB ケーブル 1	マイコンボード 1	ロボプロシールド 1	姿勢検出シールド 1
			

液晶ディスプレイシールド 1


図0-1　必要なもの

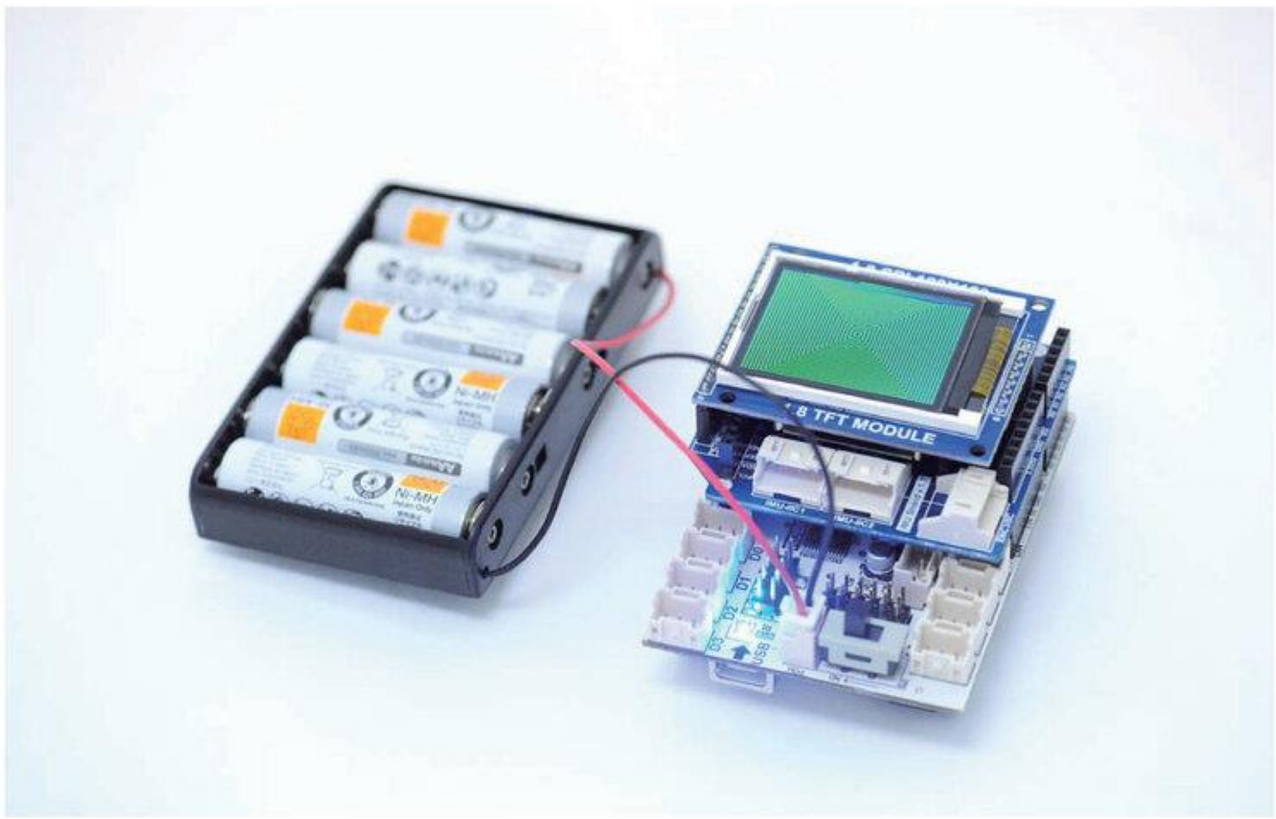


図0-2　液晶ディスプレイユニット

講　電池ボックスを使用しない場合は、ロボプロシールドの接続は不要です。

1．　デジタルの世界について（目安 20 分）

1.0.　コンピューターの仕組み

突然ですが、ものすごくごちゃごちゃに散らかってしまった部屋の片づけをするとき、どうすればより効率的にきれいにできるか考えてみましょう。
まずは、久々に手に取ったマンガをついつい読んでしまったりせず、テキパキと手を動かせる「処理の速さ」が必要です。
また、片づけのときは散らかったものを集め、「これは本棚、これは机の引き出し…」と仕分けをする人が多いと思います。仕分けたものをいったん置いておくための「空きスペースの広さ」があるほど一度にたくさんのものを仕分けられ、片づけの効率はあがります。
最後に、仕分けたものは棚や引き出し、押し入れの中などにしまいます。しまえる場所がたくさんあるほど整理しやすいですから、「収納スペースの多さ」も、なるべくほしいですね。

実は、コンピューターの性能というのも、これと同じような要素で決まるのです。
コンピューター（パソコン）を構成する主な要素に「CPU（中央演算処理装置）」、「メモリー（主記憶装置）」、「ハードディスク（補助記憶装置）」の3つがあります。これらがそれぞれ「計算処理」「処理途中のデータの一時的な保管」「処理が終了したものの収納」を担当しています。つまり、CPUの性能が「処理の速さ」、メモリーの容量が「空きスペースの広さ」、ハードディスクの容量が「収納スペースの多さ」にそれぞれ対応しているのです。

図1-0　効率的な掃除

みなさんはこれまでロボプロでたくさんのプログラムをかいてきましたが、「この2つのプログラムはかき方は異なるが、実行してみると同じ動作をする」ということが何度もありましたね。プログラムを実行するときにはメモリーの容量を使用します。当然メモリーの容量には限りがあるので、あまり容量を使わなくて済む、効率的なプログラムの方が望ましいわけです。もし無駄の多いプログラムばかり実行するはめになるとコンピューターはより大容量のメモリーを搭載しなければならず、そうなるとコンピューターがどんどん大きく、そして価格が高くなってしまうのです。

豆知識

ここでは皆さんが授業で使っているマイコンについて紹介します。マイコンは「Atmega328P」というATMEL社製のもので、パソコンに搭載されているものと違い、CPU、メモリー、がワンチップ型になっています。「Arduino IDE（統合開発環境）」でつくられたプログラムは、コンピューターが解読する機械語に変換（コンパイル）されて、このフラッシュメモリーにかき込まれます。かき込まれたデータは電源が切れても消えることはありません。

ちなみに、最近のパソコンのメモリーは4GB（ギガバイト＝4194304キロバイト）以上のものも多いですが、ロボプロのマイコンボードは32KBしかありません。したがって、マイコンのように小さな容量のメモリーを使う場合、やはり効率的なプログラムをかくことが重要なのです。

Arduino IDEは、プログラムをかき込むときに、下の画面のように「コンパイル後のスケッチのサイズ」と表示されますが、これがデータ量になります（最大容量約32KB）。プログラムをかき込むときに意識して見てみましょう。

```
コンパイル終了。

コンパイル後のスケッチのサイズ：7,060バイト（最大容量32,256バイト）
```

余談ですが、皆さんが将来パソコンを購入するときには、ぜひメモリーの容量をチェックしてみてください。メモリーが大容量なほど、一度に複数のことができるので、用途に応じて上手にパソコンを選ぶことができます。

1.1. デジタルデータとは

さて、先ほどから「メモリーの容量」という言葉が出てきますが、これは具体的にどのようなものなのか説明します。

第2回で、4つのタクトスイッチをオンにしたりオフにしたりして様々なパターンをつくったのを覚えていますか。1つのスイッチにつき「オン」「オフ」の2通りのパターンができますから、スイッチを増やしていくことでつくれるパターンの数が増えていくことがわかりますよね。4つのスイッチだと16通りのパターンに分けることができました。

表 1-0　タクトスイッチ 4 つでつくれる 16 通りのパターン

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Sw0	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	ON	ON	ON	ON	ON	ON	ON	ON
Sw1	OFF	OFF	OFF	OFF	ON	ON	ON	ON	OFF	OFF	OFF	OFF	ON	ON	ON	ON
Sw2	OFF	OFF	ON	ON	OFF	OFF	ON	ON	OFF	OFF	ON	ON	OFF	OFF	ON	ON
Sw3	OFF	ON	OFF	ON	OFF	ON	OFF	ON	OFF	ON	OFF	ON	OFF	ON	OFF	ON

スイッチを1つ増やしていくごとに、つくれるパターンの数は2倍に増えていきます。スイッチが4つなら、$2\times2\times2\times2 = 2^4 = 16$通りとなるわけです。

16パターンではひらがなどころかアルファベット26文字すら満足にデータ化できませんが、スイッチを増やしていけば対応できるようになるのです。

やってみよう！

ひらがな46文字をすべて表現できるようにしよう。タクトスイッチが最低何個あれば、46通り以上のパターンをつくれるようになるかな？

6　個

講　$2^5=32$, $2^6=64$なので、パターン数がはじめて46を超えるのはスイッチが6つのときです。

ところで、コンピューターの世界ではスイッチがオンになっていることを「1」、オフになっていることを「0」と表現します。たとえばひらがなを「『あ』は000000、『い』は000001、…」とそれぞれスイッチ6個ずつで表現できるようにすると「こんにちは」という文は「001001（こ）101101（ん）010101（に）010000（ち）011001（は）」と30個分のスイッチで表すことができます。人間の私たちにとっては回りくどく、しかも大変な方法としか思えませんが、コンピューターにとってはこのやりかたの方が合っているのです。

コンピューターは、ものすごく乱暴（らんぼう）にたとえると「数えきれないほどたくさんのスイッチを、とんでもないスピードでオン・オフする機械」と言えます。ロボプロで使われているコンピューター（マイコン）は理論上、1秒間にだいたい1億3000万以上のスイッチを処理（しょり）できます。コンピューターの考え方は人間からすると回りくどいですが、そのスピードは想像を超（こ）える速さなのですね！

ちなみに、スイッチ１個分のデータ量のことを「1ビット（bit）」とよびます。先ほどあげた「こんにちは」のデータは全部でスイッチ30個分でしたから、30ビットのデータという事になります。

1.2.　画像のデジタルデータ

さて、コンピューターが扱うデジタルデータはなにもひらがなどの「文字」だけではありません。音楽や画像、映像といったデータも、当然0と1の組み合わせでできています。第1回でも簡単に説明しましたが、液晶ディスプレイに画像やオリジナルのキャラクターを表示させたいときに役立つ知識なので、もう少し詳しく知っておきましょう。

説明をわかりやすくするため、下のような9マス分の枠(わく)をつくり、それぞれのマスを白か黒のどちらかに塗(ぬ)りつぶしたシンプルな画像を考えてみましょう。

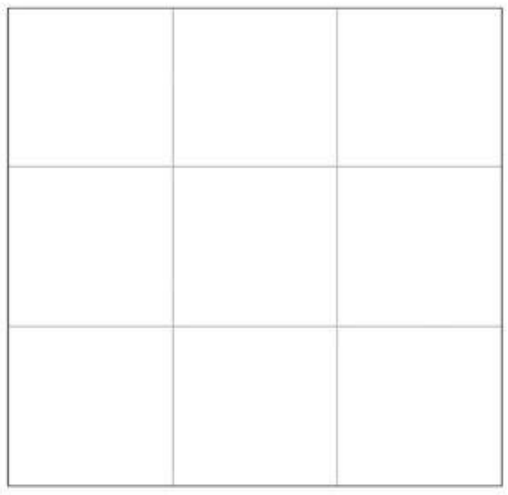


図1-1　9マス分の画像

白か黒の2通りに分ければいいだけなので、1マスにつき1ビットあれば十分です。白く塗(ぬ)るマスを0、黒く塗(ぬ)るマスを1と表すことにしましょう。

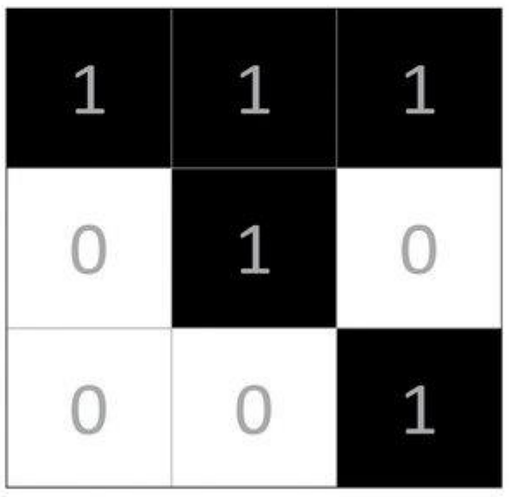
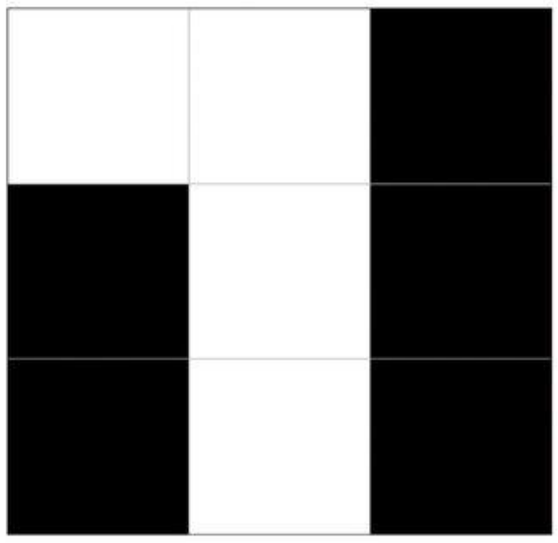


図1-2　1ビットの白黒画像

上の画像であれば、左上のマスから順に数字を並べていくと「111010001」となります。つまり画像を9ビットのデータにすることができました。

やってみよう！

以下の画像も同様にデータ化してみよう！



001101101

次に、カラー画像に挑戦してみましょう。
第1回、第2回で何度か扱いましたが、液晶ディスプレイにかく線や図形の色は、赤（R)、緑（G)、青（B）の3色をどれくらい混ぜるかで色を指定していました。
今回はひとまず、1マスにつき「赤で塗る or 塗らない」「緑で塗る or 塗らない」「青で塗る or 塗らない」という3つの情報を入れるようにしましょう。たとえば「赤は塗る、緑は塗らない、青は塗らない」というマスであれば「100」と表せます。1マスあたり3ビットになりますね。

図1-3　3ビットカラー画像（三原色）

3ビットなので、2×2×2=8通りのパターンをつくることができます。たとえば「101」なら「赤は塗る、緑は塗らない、青は塗る」なので、赤と青を混ぜて紫色になりますね。8通りの色とそのデータを以下にまとめます。

図1-4　3ビットカラー画像（色見本）

では、「3ビットカラー画像」のデータをつくってみましょう。

やってみよう！

以下のカラー画像は、データ化するとどんな文字列になるかな？
左上から横にデータ化していこう。

101001111011110010001000100

講　紫は101、青は001、白は111、…と並べていきます。

27ビットのデータになりましたね。それぞれの色に割り当てるビット数を増やしていけば、「濃い赤とやや濃い緑と薄い青を混ぜる」などもできるようになるため、つくれる色がどんどん増えていくわけです。
ロボプロのプログラムでは基本的に赤緑青それぞれに8ビットずつを割り当て、0 ～ 255の256通りの濃さで色を塗りわけられるようになっています。
`TFTscreen.stroke(255,255,255);`などといったかき方をしていましたね。なお、この場合は赤も緑も青も255、つまり一番濃い色で塗るという命令なので、3ビットの「111」と同じく白になります。

1.3.　16 進数

いま説明したとおり、コンピューターのデジタルデータは0と1のみで構成された「2進数」でできています。しかし、ひらがな1文字表すのに数字が6けたも必要だったりと、データ化すると人間の目にはかえってごちゃごちゃしているように見えます。
値が多くなってしまうのは、2進数がその名の通り「数字を2つ使ったら次のケタに進まなければいけない」ためです。私たちが普段使っている「10進数」であれば、0 ～ 9の10個を使ってようやく次のケタですから、それと比べると2進数データのケタ数がふくれ上がってしまうのも無理のないことです。
とはいえ、やはりケタ数が多いデータは見るのもかくのも大変です。そのため、プログラミングの際には「2進数よりも1ケタあたりに使える数字がたくさんある」方式が使われることが多いのです。ここで登場するのが「0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F」の16個の「数」を使って次のケタに進む「16進数」です。Fの次は「10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 1F, 20, 21, 22, …」と続いていきます。
なぜ10進数でなく16進数を使うのかというと、2進数との相性がいいからです。
2進数で8ビットのデータは、00000000から11111111まで合わせて256通りつくれます。また、16進数で2けたの数をつくると、00 ～ FFまで256通りつくれます。つまり、2進数8ケタ分のデータが、16進数だとちょうど2けたに収められるというわけです。
液晶ディスプレイのプログラムをかく際、この16進数をたびたび使います。特に、10進数の「0, 128, 255」が16進数だと「00, 80, FF」となることは覚えておきましょう。

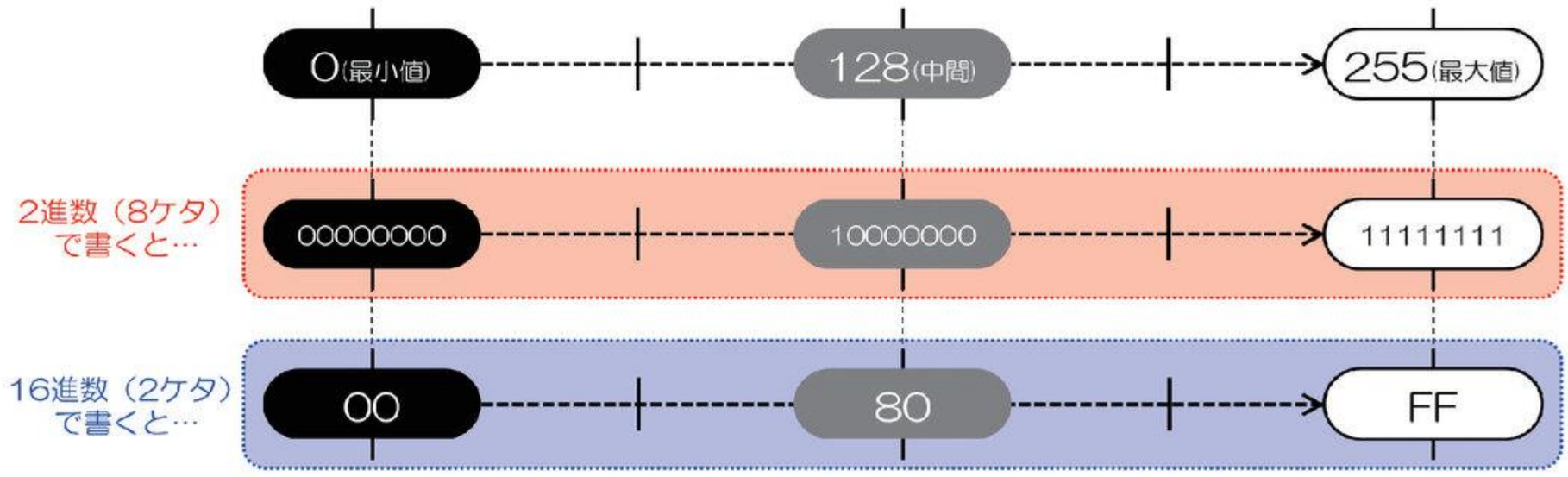


図1-5　10進数と2進数、16進数の関係

コラム　バイトと補助単位

先ほどデータ化したカラー画像ですが、たったの9マス・8色でも27ビットでした。しかしロボプロで使用している液晶ディスプレイは、第1回でも説明したとおり2万以上のマスを持っています。最新式のデジタルカメラともなれば、その1千倍、つまり2千万以上のマスを、それぞれ1,600万色以上の色で塗(ぬ)りわけることもできます。画像データひとつでいったいどれほどのビット数になるか想像もつかないですね。
このように、コンピューターの発展とともに、扱われるデータはどんどん大きくなってきました。ビットで表すと数字がとても大きくなってしまい不便なので、さまざまな工夫がなされています。

まず、コンピューターが開発された英語圏ではアルファベット（大文字・小文字）、数字、「!」や「&」といった一般的な記号などをすべて合わせると表しきるのに7～8ビット必要でした。いわば「8ビットが1文字分のデータ量」ともいえる状況だったのです。そのため、8ビットをひとまとめにして「1バイト（byte）」とする新たな単位ができました。
さらに、バイトはg（グラム）やm（メートル）といった他の単位と同様に、値が大きくなったら「k（キロ）」をつけてひとまとめにできるよう決められました。
グラムやメートルは1,000まとめると「キロ」がつきますが、バイトだけは1,024まとめると「キロ」になります。1キロバイト（KB）=1,024バイト=8,192ビットということになります。これで、値をかなり小さくできますね。ちなみに、他の単位につく「キロ」とは厳密にいうと別物なので、小文字ではなく大文字の「K」で表されることが多いです。
これでも不十分なときは、1,024キロバイトをまとめて1メガバイト（MB）、1,024メガバイトをまとめて1ギガバイト（GB）、…としていきます。キロ、メガ、ギガといったものを「補助単位」とよびます。

1バイト	（B）	＝1ビット（b）×8
1キロバイト	（KB）	＝1バイト×1024
1メガバイト	（MB）	＝1キロバイト×1024
1ギガバイト	（GB）	＝1メガバイト×1024
1テラバイト	（TB）	＝1ギガバイト×1024
1ペタバイト	（PB）	＝1テラバイト×1024
1エクサバイト	（EB）	＝1ペタバイト×1024
1ゼタバイト	（ZB）	＝1エクサバイト×1024
1ヨタバイト	（YB）	＝1ゼタバイト×1024

最近ではスマートホンのデータ通信量などは「ギガバイト（GB）」で表すことが多いのではないでしょうか。パソコン用のハードディスク容量などでは「テラバイト（TB）」が使われることもあります。
とはいえコンピューター技術はどんどん発展しています。「SDカード」とよばれる記録メディアは、1999年に開発された当初は8MB程度の容量しかありませんでしたが、いまでは数百GBのものも一般的です。
皆さんが大人になるころには、ひょっとしたら「128エクサバイト程度のメモリーカードでは容量不足だな」とか「先月のスマートホンの通信量が40ペタバイトを超えてしまった」といった話が出てくるかもしれませんね。

2.　液晶ディスプレイにキャラクターをかく（目安 50 分）

2.0.　データ型の変数

先ほどまでの部分で、「画像データをつくるためには、コンピューターにどのような命令をすればいいか」がなんとなくわかってきたのではないでしょうか。
では、今度はもっとマスや色の数を増やして、より複雑（ふくざつ）な画像データをつくりましょう。
最終的にはゲームのキャラクターデータとして使用したいので、以下のような画像をめざしてみます。

図2-0　「たこ？」の画像データ

さっそくプログラミングといきたいところですが、まずは**表2-0**を見ておきましょう。ロボプロのプログラムでは、変数をつくるときほとんどが int、ごくまれに float という種類（型）を使うばかりで、それ以外の変数型と出会うことがあまりありませんでした。今回、あるいは今後プログラミングをしていくうえでも重要な知識なので、これを機にしっかり知っておきましょう。

表2-0　変数のデータ値

	型名	バイト数	扱える数値の範囲（はんい）
整数型	int	2バイト	−32768 ～ 32767
	long	4バイト	−2147483648 ～ 2147483647
	unsigned int	2バイト	0 ～ 65535
	unsigned long	4バイト	0 ～ 4294967295
	byte	1バイト	0 ～ 255までの8ビットの数値を格納します。
浮動（ふどう）小数点型	float	4バイト	$-3.4*10^{38}$ ～ $3.4*10^{38}$
	double	4バイト	floatと同じ
データ型	char	1バイト	一つの文字を記憶するために1バイトのメモリーを消費する型です。 −128 ～ 127
	unsigned char	1バイト	符号なしのデータ型で１バイトのメモリーを消費する型です。 0 ～ 255

型によって、消費するビット数（バイト）が違（ちが）い、使用できる値が違（ちが）います。割り振りによってできることが違（ちが）うということです。「だったら大きい方がいいじゃないか！」と思う人もいるかもしれませんが、条件によっては、サッカー場でフットサルを行うような状況が発生して、メモリーがもったいないですよね。使い道にあったことをメモリーに行わせることが大切です。一般的には、「 char 」は文字列に使用したり、「 int 」は計算に使用したりします。2バイトで多くの計算はできますよね。「 float 」、「 double 」は小数点の演算に使用しますので、数学的な計算には必要になります。

2.1. 白黒画像をかく

まずは、以下のプログラムを実行してください。

プログラムの書き込み

RoboticsProfessorCourse3　＞　MagicItemLCD3　＞　Line1

実行結果：液晶ディスプレイに斜(なな)めの線が表示される。
第1回でも直線をかく命令TFTscreen.line();を学びましたが、今回はそれとはちがった方法で直線をかいています。

プログラム「Line1」より抜粋(ばっすい)

```
void setup(void){
    TFTscreen.begin();                    //液晶ディスプレイ表示
    TFTscreen.background(0, 0, 0);        //背景色は黒
    TFTscreen.stroke(255, 255, 255); //線の色は白
    for(int i = 0; i < 100; i++){
        TFTscreen.point(i, i);
    }
}
```

命令「TFTscreen.point」
実行結果：指定された座標に点をかく
使 い 方：TFTscreen.point([x 座標], [y 座標]);

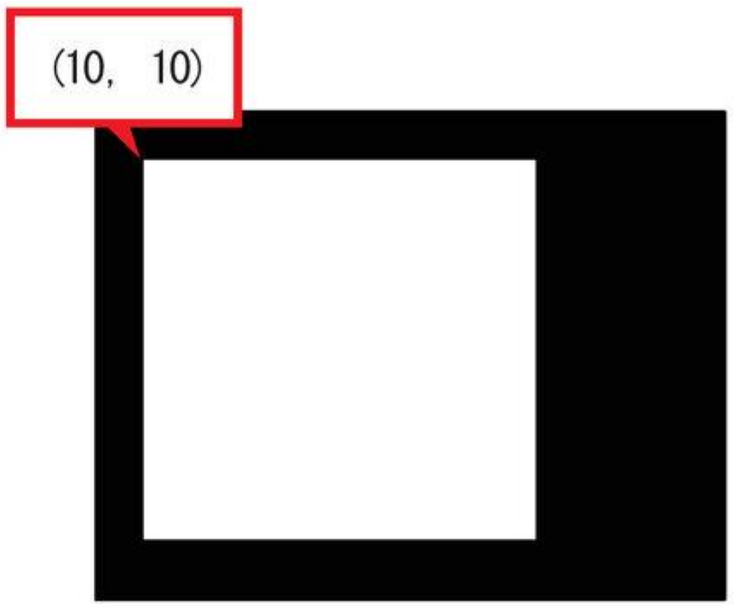
for文を使い、(0, 0)、(1, 1)、(2, 2)、…と一つひとつ点をかくことで直線に見せています。ただ直線をかきたいだけなら、当然 TFTscreen.line(); の方が簡単ですし、メモリーも消費しません。しかし、これまで学んできた TFTscreen.line(); や TFTscreen.rect();、TFTscreen.circle();とちがって、この命令は決まった図形以外を自由にかくことができるので、今回のような場合に役立ちます。
ただし、1回の命令で1つの点しかかけないという特性上、for文などのくり返し命令が必要不可欠です。少し、for文を使った描画を練習しておきましょう。

やってみよう！

プログラム「Line1」をかきかえ、TFTscreen.pointとfor文を組み合わせて以下の図形をかくプログラムにしてみよう！

① (10, 10)からx軸方向に延びる直線

② (10, 10)を一番左上の頂点とする正方形



講

解答例はそれぞれ以下の通りです。

①

```
for(int i = 0; i < 100; i++){
    TFTscreen.point(i + 10, 10);
}
```

②

```
for(int j = 0; j < 100; j++){
    for(int i = 0; i < 100; i++){
        TFTscreen.point(i + 10, j + 10);
    }
}
```

さて、次は以下のような三角形をかく方法を考えてみましょう。

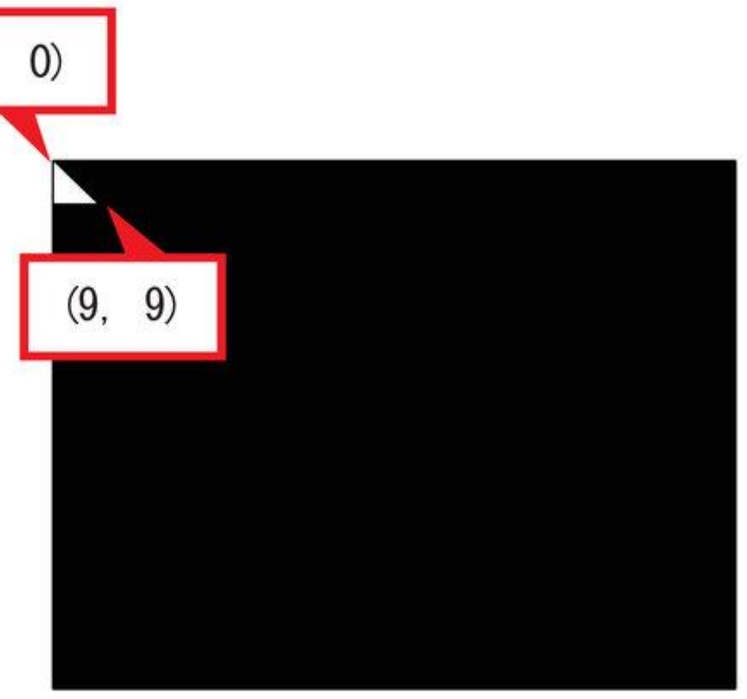


図2-1　point命令で三角形をかく

一番上の行は`i < 1`、一つ下の行は`i < 2`、その下の行は`i < 3`、…と、終わりの条件が行ごとにちがいますね。for文をさらに増やしてかいてもいいのですが、以下のように考えられるようになると今後役立ちます。

図2-2　画像を四角形ととらえる

全体はあくまで「四角形」ととらえ、白で塗(ぬ)るマスと黒で塗(ぬ)るマスを分けるという発想です。
そしてこの図、どこかで見たような気がしませんか？
「さっき0と1でデータ化した図に似ているな」と思う事ができた人は大正解です！
白を1、黒を0とすると以下のようなデータになりますよね。

```
1000000000110000000011100000001111000000111110000011111100001111111000111111111100111111111111011111111111
```

実は、先ほど学んだこのデータ化が、三角形描画の大きなヒントになります。

プログラム「Line1」の`void setup()`の上に、以下のような文を追加してみましょう。

```
const char chara[] = {
      1, 0, 0, 0, 0, 0, 0, 0, 0, 0
    , 1, 1, 0, 0, 0, 0, 0, 0, 0, 0
    , 1, 1, 1, 0, 0, 0, 0, 0, 0, 0
    , 1, 1, 1, 1, 0, 0, 0, 0, 0, 0
    , 1, 1, 1, 1, 1, 0, 0, 0, 0, 0
    , 1, 1, 1, 1, 1, 1, 0, 0, 0, 0
    , 1, 1, 1, 1, 1, 1, 1, 0, 0, 0
    , 1, 1, 1, 1, 1, 1, 1, 1, 0, 0
    , 1, 1, 1, 1, 1, 1, 1, 1, 1, 0
    , 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
};
```

要はchar型変数を、chara という名前で使うという宣言です。int i = 0;などの宣言と同じような形です。 [ ]をつけているため、変数chara には配列が用いられていることがわかります。
カンマで区切り、見やすいように改行を入れた形になっていますが、数字の並びは先ほどデータ化したものとまったく同じです。つまり、変数chara[] の値は場合によって0になったり1になったりするということです。
これを用いると、変数 chara[] は1マス目を塗るときはchara[0] = 1に、2マス目を塗るときはchara[1] = 0に、3マス目を塗るときはchara[2] = 0に、…とそれぞれ置きかわります。これをうまく利用して、1マス塗るごとに塗る色を黒、または白で指定できるようにすれば三角形がかけます。

ステップアップ

プログラム「Line1」をかきかえ、三角形をかくプログラムを完成させよう！

ヒント

変数chara[] が0だったときはTFTscreen.stroke(0, 0, 0);、1だったときはTFTscreen(255, 255, 255);で塗られるようにすればいいんだね！
以下の表は、それぞれのマスの変数 chara[] の対応を示しているよ。赤丸のマスを塗るときは、変数 chara[36] を参照するようにすればいいということになるね。

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	10	11	12	13	14	15	16	17	18	19
2	20	21	22	23	24	25	26	27	28	29
3	30	31	32	33	34	35	36	37	38	39
4	40	41	42	43	44	45	46	47	48	49
5	50	51	52	53	54	55	56	57	58	59
6	60	61	62	63	64	65	66	67	68	69
7	70	71	72	73	74	75	76	77	78	79
8	80	81	82	83	84	85	86	87	88	89
9	90	91	92	93	94	95	96	97	98	99

どうでしょうか。今までの課題から見ても、今回はかなり難しいです。
解答例は以下の通りです。

```
for(int j = 0; j < 10; j++){
    for(int i = 0; i < 10; i++){
        TFTscreen.stroke(255 * chara[10 * j + i], 255 * chara[10 * j + i],
255 * chara[10 * j + i]);
        TFTscreen.point(i, j);
    }
}
```

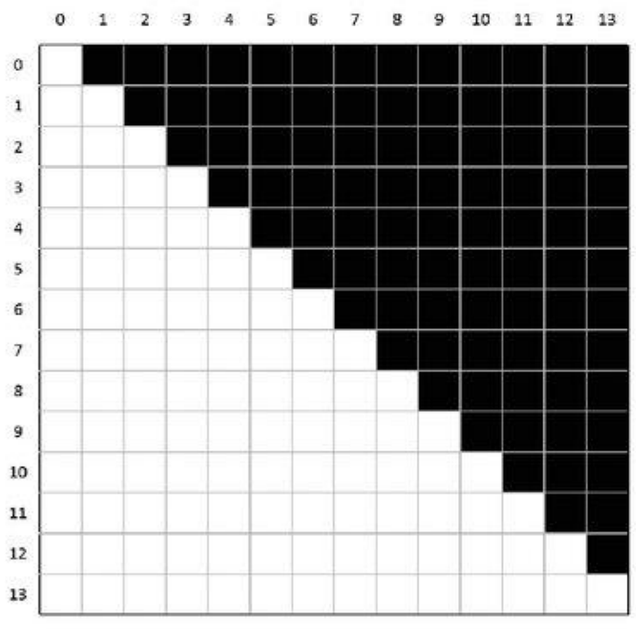
1マスごとに色をかえる必要があるため、TFTscreen.stroke();はfor文の中に必要ですね。255 * chara[]というかき方をすることで、変数 chara[]が0のときはRGBの値が255×0=0、1のときは255×1=255となり白と黒のどちらかで塗(ぬ)ることになります。

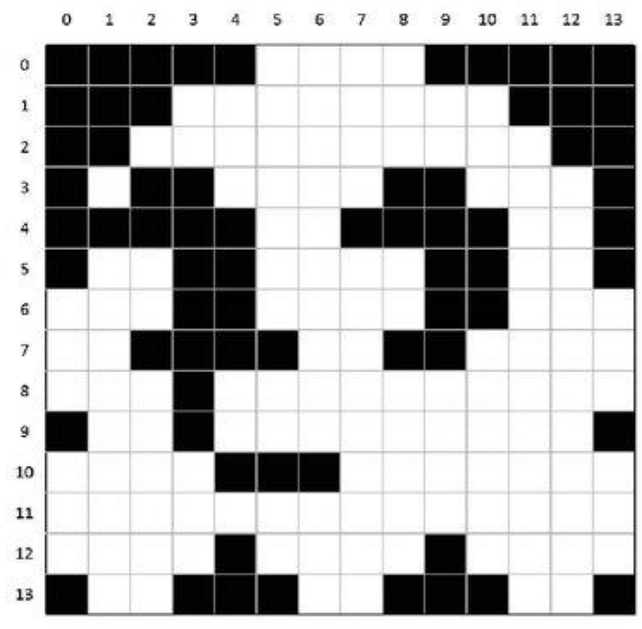
そして、たとえばi = 6でj = 3のマスを塗(ぬ)るときは、chara[36]を用いることになります。10 * j + iというかき方をすることで、「36」という2けたの数をつくり出せるようになっています。中学校の数学でも、似たような問題をやったことがあるのではないでしょうか？

これで、円や三角形といった基本図形に限らず、あらゆる形の物体をかくことができるようになりましたね！
いまは練習のために縦(たて)・横10マスずつの三角形でしたが、当然配列の中のデータを増やしていくことで、もっと大きな図形をかくこともできます。どんどん練習していきましょう。

チャレンジ課題

先ほどつくったプログラムをさらにかきかえ、以下のような図形を表示させてみよう。
① 先ほどかいた三角形を、縦(たて)・横ともに14マスにしたもの。
② ゲームキャラクターの白黒バージョン（縦(たて)・横ともに14マス）





講　解答例は巻末に記載します。

2.2.　カラー画像をかく

さて、今回学んできた知識をフル活用すると、液晶ディスプレイに表示する画像をカラーにすることもできます！
ためしに、以下のプログラムを実行してみましょう。

プログラムの書き込み

RoboticsProfessorCourse3　＞ MagicItemLCD3　＞ DotChara

実行結果：以下の画像が液晶ディスプレイに表示される。

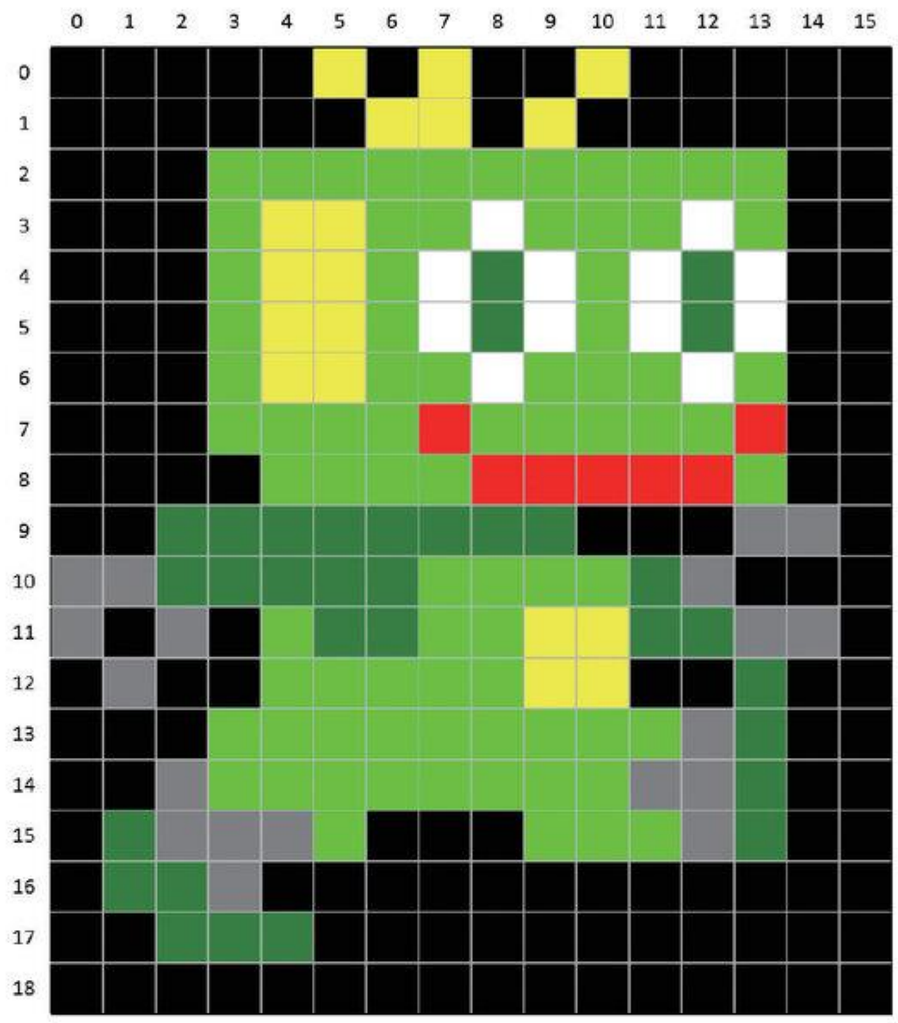


図2-3　表示されるキャラクター画像

カラーのキャラクターになりました。先ほど白黒画像をかいた時と、プログラムのつくりはそれほど変わりません。確認してみましょう。

プログラム「DotChara」より抜粋（ばっすい）

```
//キャラクターの大きさ指定
#define sizeX 16                    //キャラクターのドットサイズ横
#define sizeY 19                    //キャラクターのドットサイズ縦

//色見本データ
const char R[] = {
    0x00, 0x00, 0xff, 0xff, 0x00, 0x00, 0xff, 0xff, 0x80, 0x00, 0x80, 0x80,
0x00, 0x00, 0x80, 0xc0
};
const char G[] = {
    0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0x80, 0x00, 0x00, 0x00,
0x80, 0x80, 0x80, 0xc0
};
const char B[] = {
    0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x80, 0x80, 0x00, 0x80,
0x00, 0x80, 0x00, 0xc0
};

//キャラクターデータ
const char chara[] = {
    0, 0, 0, 0, 0, 6, 0, 6, 0, 0, 6, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 6, 6, 0, 6, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 0, 0,
    0, 0, 0, 4, 6, 6, 4, 4, 7, 4, 4, 4, 7, 4, 0, 0,
    0, 0, 0, 4, 6, 6, 4, 7,12, 7, 4, 7,12, 7, 0, 0,
    0, 0, 0, 4, 6, 6, 4, 7,12, 7, 4, 7,12, 7, 0, 0,
    0, 0, 0, 4, 6, 6, 4, 4, 7, 4, 4, 4, 7, 4, 0, 0,
    0, 0, 0, 4, 4, 4, 4, 2, 4, 4, 4, 4, 4, 2, 0, 0,
    0, 0, 0, 0, 4, 4, 4, 4, 2, 2, 2, 2, 2, 4, 0, 0,
    0, 0,12,12,12,12,12,12,12,12, 0, 0, 0, 8, 8, 0,
    8, 8,12,12,12,12,12, 4, 4, 4, 4,12, 8, 0, 0, 0,
    8, 0, 8, 0, 4,12,12, 4, 4, 6, 6,12,12, 8, 8, 0,
    0, 8, 0, 0, 4, 4, 4, 4, 4, 6, 6, 0, 0,12, 0, 0,
    0, 0, 0, 4, 4, 4, 4, 4, 4, 4, 4, 4, 8,12, 0, 0,
    0, 0, 8, 4, 4, 4, 4, 4, 4, 4, 4, 8, 8,12, 0, 0,
    0,12, 8, 8, 8, 4, 0, 0, 0, 4, 4, 4, 8,12, 0, 0,
    0,12,12, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0,12,12,12, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};

void loop(){
    for(int y = 0; y < sizeY; y++){          //縦方向のデータを描画する
        for(int x = 0; x < sizeX; x++){      //横方向のデータを描画する
        TFTscreen.stroke(B[chara[y * sizeX + x]], G[chara[y * sizeX +x]],
R[chara[y * sizeX + x]]);                    //色見本データを参照する
        TFTscreen.point(100 + x, 50 + y); //データをかく
        }
    }
}
```

まず、画像の縦(たて)、横のマスの数を、数字ではなく sizeX 、sizeY という文字列で指定するようにしました。プログラムのはじめにそれぞれの値を指定する #define が追加されています。

最大のちがいは変数 chara の配列部分でしょう。0と1の2種類ではなく、0、2、4、6、7、8、12と多種多様な数字が使われるようになっています。また、各マスを塗(ぬ)るときの色の指定も、先ほどの白黒画像とは少しだけ異なります。

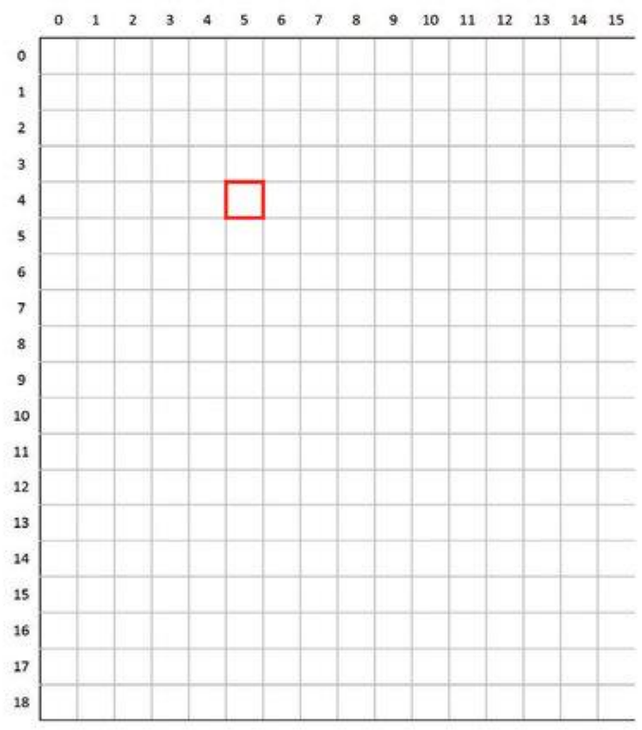
プログラム「DotChara」より抜粋(ばっすい)

```
TFTscreen.stroke(B[chara[y * sizeX + x]], G[chara[y * sizeX + x]],
R[chara[y * sizeX + x]]);
```

白黒画像のプログラムとちがって、赤緑青にそれぞれ専用の変数R[ ]、G[ ]、B[ ]を使うようにすることで、さまざまな色がつくれるようになっています。しかし、[ ]の中にさらに[ ]が入っていたり、さらにその中も変数だらけになっていたりと、非常に難しいプログラムになっています。

しっかりと理解するため、一つひとつゆっくりと読み解いていきましょう。

やってみよう！



ためしに、枠(わく)のマスを塗(ぬ)るとき、どの変数にどのような値が入るか考えていくよ！

① 赤枠(わく)マスのx座標、y座標と、sizeX の値はハッキリしているね。まずはこの3つの値を代入し計算しよう！

TFTscreen.stroke(B[chara[y * sizeX + x]], G[chara[y * sizeX + x]], R[chara[y * sizeX + x]]);

TFTscreen.stroke(B[chara[69]], G[chara[69]], R[chara[69]]);

② ①で chara[69] になることはハッキリしたよね！　プログラムをよく読んで、chara[69] の値はいくつになるか探してみよう！

6

これで、色の指定部分が以下のようになるところまではわかりましたね。

```
TFTscreen.stroke(B[chara[y * sizeX + x]], G[chara[y * sizeX + x]], R[chara[y * sizeX + x]]);
```

▼

```
TFTscreen.stroke(B[        6        ]], G[        6        ]], R[        6        ]);
```

あとは、B[6]、G[6]、R[6]がそれぞれいくつになるのかわかるようになれば、カラー画像のかき方もわかると思います。

ステップアップ

プログラム「DotChara」をジックリと読み、以下の質問に答えよう！

① B[6]、G[6]、R[6] の値はそれぞれいくつになるかな？　プログラム内から探し出し、抜き出してみよう！

　また、このRGB値は何色になるかな？

B[6]：0x00　　G[6]：0xff　　R[6]：0xff　　色：黄

② 変数 chara の配列内で「0」「2」「4」「7」「8」「12」とされている部分も、RGBがそれぞれいくつになるか読み取ってみよう！

0：R　0x00　　G　0x00　　B　0x00

2：R　0xff　　G　0x00　　B　0x00

4：R　0x00　　G　0xff　　B　0x00

7：R　0xff　　G　0xff　　B　0xff

8：R　0x80　　G　0x80　　B　0x80

12：R　0x00　　G　0x80　　B　0x00

読み取れましたか？

今回はマスの塗(ぬ)りかただけではなく、色のデータも配列にしてかかれていましたね。

プログラム「DotChara」より抜粋(ばっすい)

```
//色見本データ
const char R[] = {
    0x00, 0x00, 0xff, 0xff, 0x00, 0x00, 0xff, 0xff, 0x80, 0x00, 0x80, 0x80,
0x00, 0x00, 0x80, 0xc0
};
const char G[] = {
    0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0x80, 0x00, 0x00, 0x00,
0x80, 0x80, 0x80, 0xc0
};
const char B[] = {
    0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x80, 0x80, 0x00, 0x80,
0x00, 0x80, 0x00, 0xc0
};
```

0x●●とプログラム内にかかれた部分は、●●の部分を16進数として扱うことを表しています。たとえば、0xffは16進数で「FF」、つまり10進数でいう255であるという意味です。今回の色見本データは赤緑青それぞれ00、80、FFを中心に色をつくっています。値が大きいほど明るい色、小さいほど暗い色になります。全部で16個の色データがありますので、以下にまとめます。

表2-1　色番号

色番号	色	色番号	色
0	黒	8	灰色
1	明るい青	9	少し暗い青
2	明るい赤	10	少し暗い赤
3	明るい紫	11	少し暗い紫
4	明るい緑	12	少し暗い緑
5	明るい水色	13	少し暗い水色
6	明るい黄色	14	少し暗い黄色
7	白	15	少し暗い白

豆知識

プログラム「DotChara」のキャラクターデータ部分は1ケタの数と2ケタの数が混在しており、0と1だけでかけていた頃と比べると少し見づらくなってしまっています。このような問題も、16進数を取り入れることで解決できます。以下のプログラムを見てみましょう。

RoboticsProfessorCourse3　＞ MagicItemLCD3　＞ DotCharaR

表示される画像は「DotChara」と同じですが、キャラクターデータの部分に16進数が使われているのがわかりますか？

16進数であれば、0～15の数をすべて同じケタ数で表すことができるため、比較的見やすくプログラムをかくことができます。

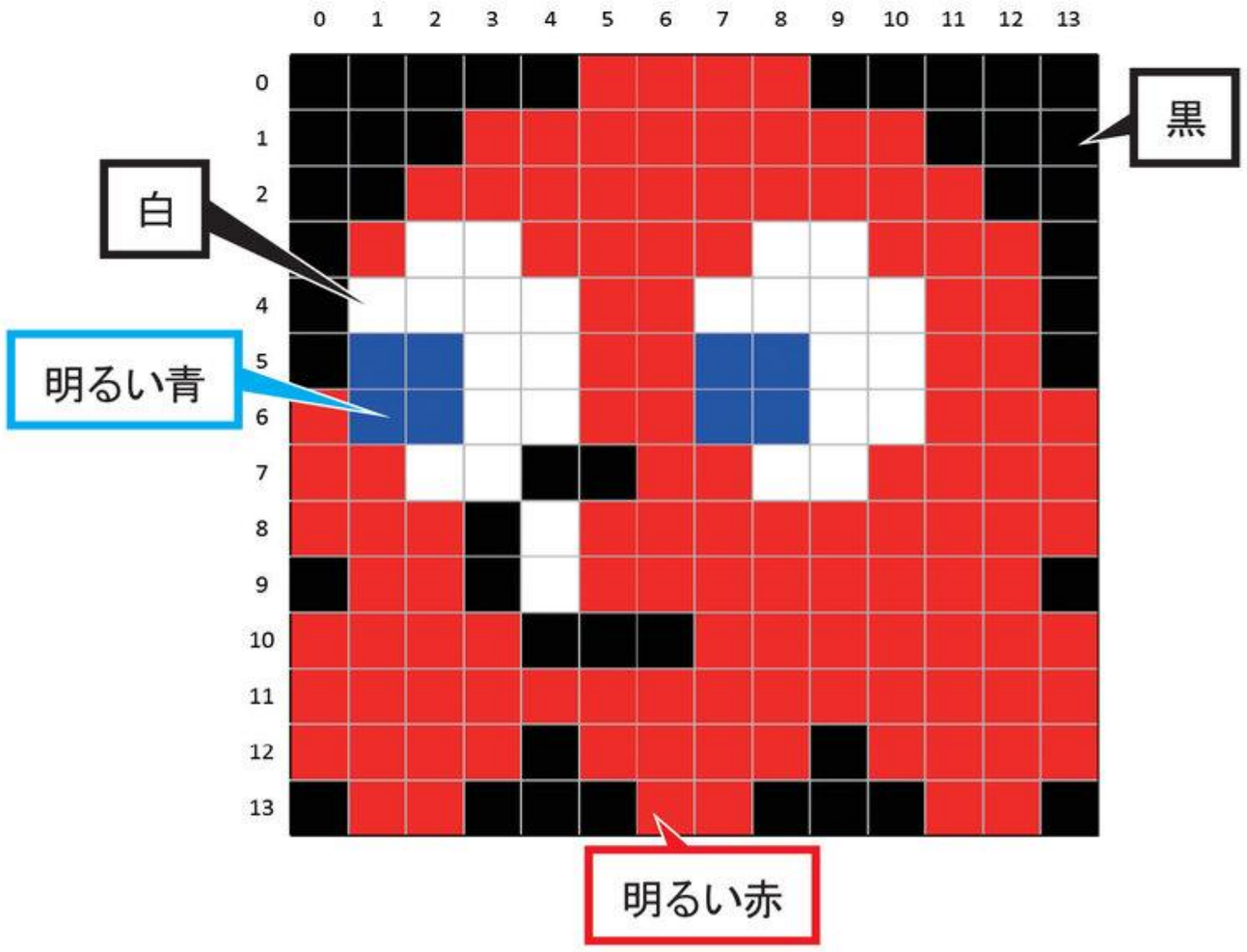
「動けばなんでもいいや」から「なるべく効率的で見やすいプログラムにしよう」へ変化できると、プログラマーとしてのレベルは大きくアップします。ぜひ覚えておきましょう。

それでは、最終ミッションです。はじめに見た「たこ？」の画像をつくってみましょう。

ステップアップ

プログラム「DotChara」をかきかえ、「たこ？」の画像データを完成させよう！

詳しい色配置は以下の通りだよ。



講

解答例は以下のプログラムです。

RoboticsProfessorCourse3　＞ MagicItemLCD3　＞ DotChara3

なお、上記プログラムではキャラの描画位置を(100, 50)としていますが、この措置は必須ではありません。

3.　ビット演算（目安 15 分）

3.0.　ビット演算とは

さて、液晶ディスプレイにキャラクターを表示させる方法を学びました。
ところで、前半に勉強した「デジタルデータ」の知識を、もう少し深めてみましょう。
赤緑青1ビットずつの3ビットカラー画像データを扱いましたね。これを例にします。

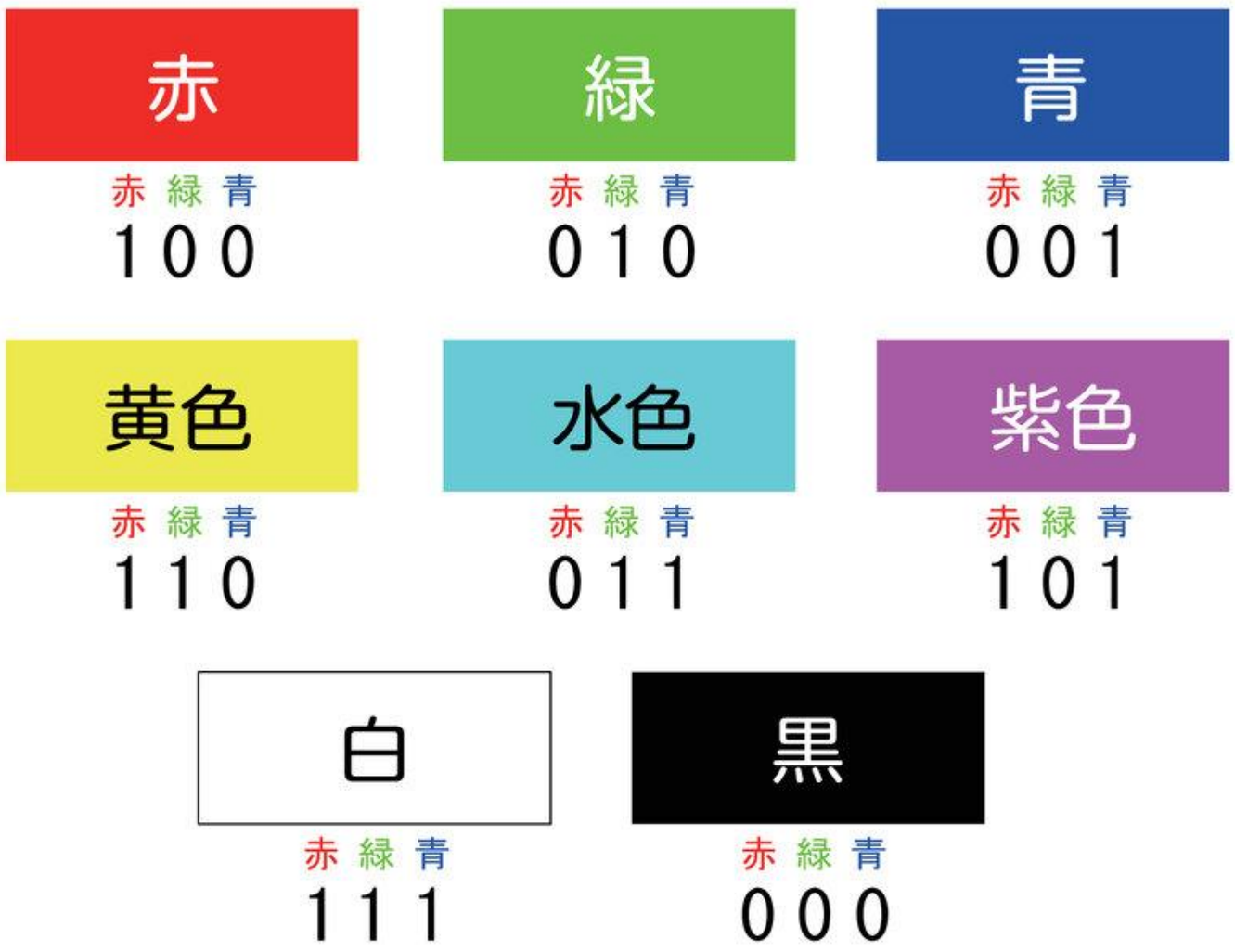


図3-0　3ビットカラー画像の色見本

赤と青を混ぜると紫色（むらさき）になりますが、これをデータ上で考えてみましょう。

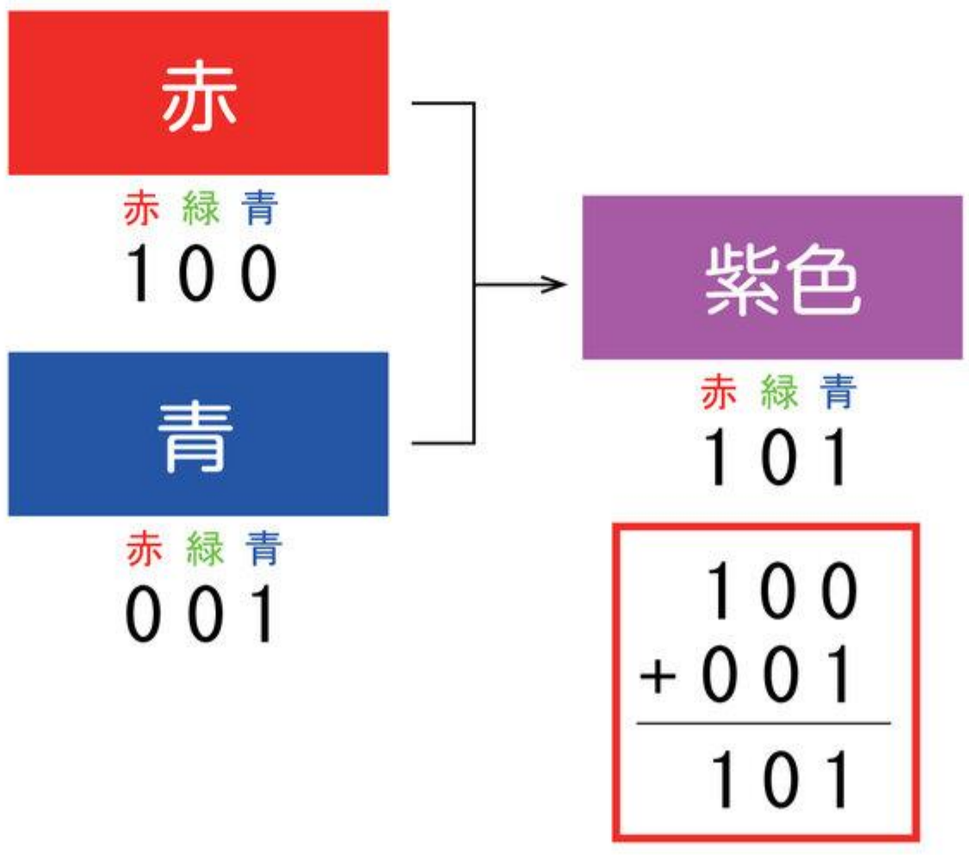


図3-1　赤と青を混ぜるときのデータ計算

赤が100、青が001ですから、100+001=101です。これは感覚的にわかりやすいですね。

では、緑と緑を混ぜるとどうなるか、考えてみましょう。緑は010ですから、010+010ですね。
いつもの感覚で10進数を使うなら、010+010＝020と言いたいところですが、そもそもこのデータは2進数でできていますから「2」を使うことはできません。
では、10進数に戻して足してみましょう。「010」という2進数は、10進数では「2」になりますから、2+2＝4です。これを2進数に戻せばいいでしょうか。
「4」を2進数で表すと「100」です。つまり、010+010＝100ということになります。色に戻して考えてみましょう。

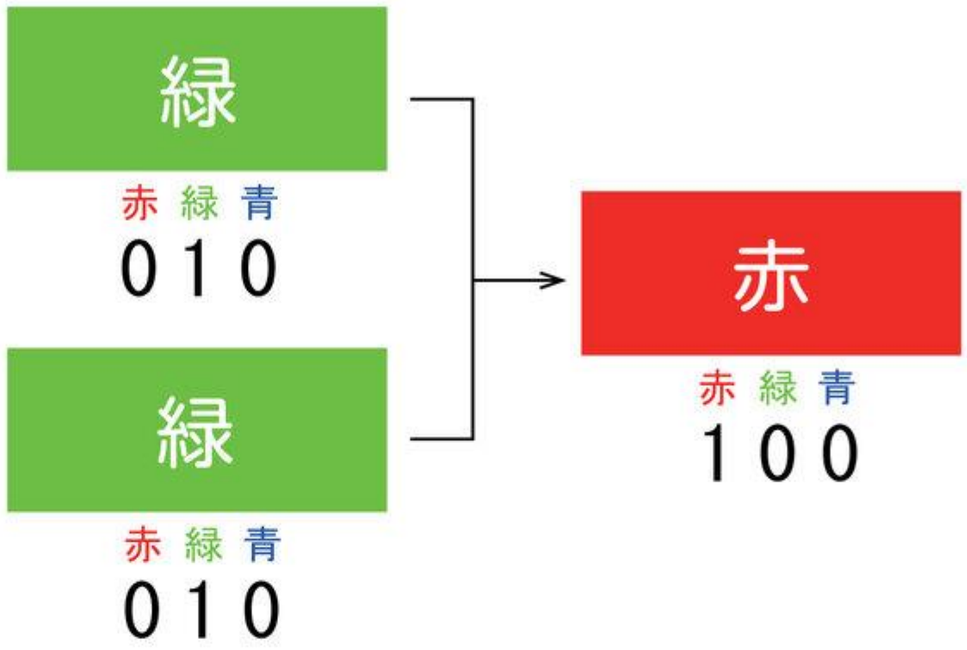


図3-2　緑と緑を混ぜるとどうなるか

「100」は赤のデータです。このままだと、緑と緑を混ぜたら赤になる、という話になってしまいますから、どうもこの計算もちがっているようですね。

実際は緑と緑を混ぜても緑のままですから、正しくは以下の図のようになります。

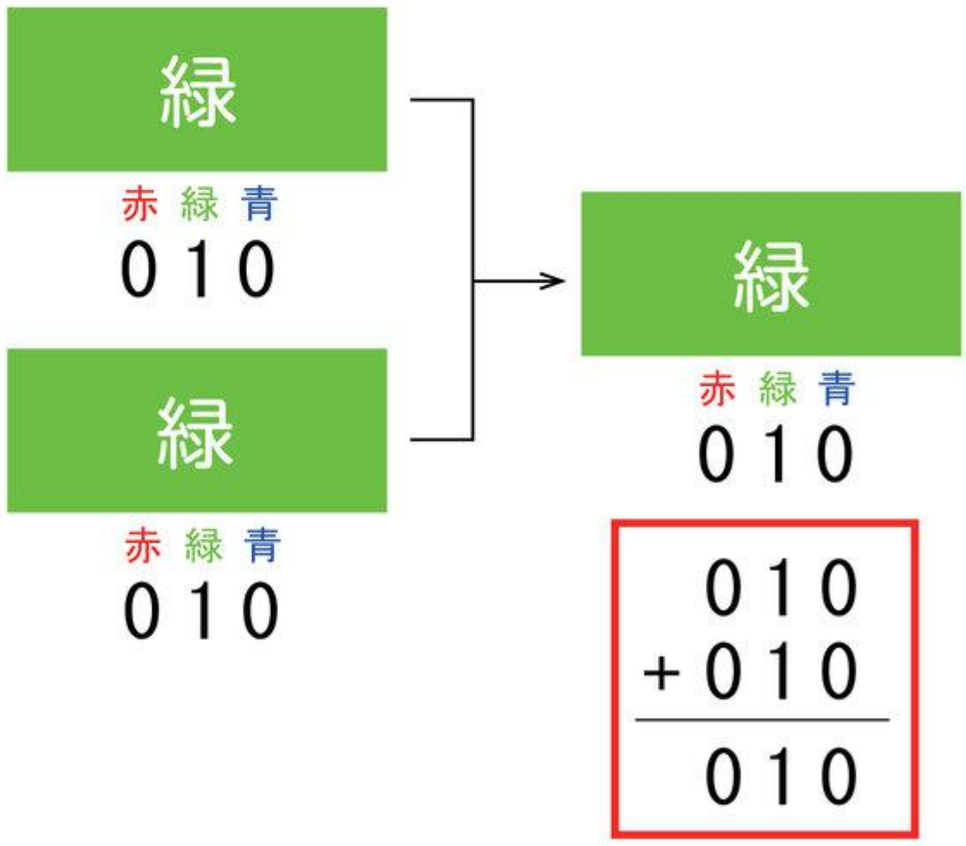


図3-3　緑と緑を混ぜるとどうなるか（正解）

010+010＝010という、いつもの足し算から考えると明らかにおかしい計算になっていますが、これが正解ですよね。このデータは左の位、真ん中の位、右の位にそれぞれ別の色のデータが入っているので、くり上がりなどで他の位を変化させることはできないのです。

コンピューター上のデータというのはこのように、位ごとに独立した情報が入っていることが多いです。そのため、複数のデータを合わせるときには同じ位どうしだけを見て、くり上がりなどで他の位に影響を与えない、という特殊（とくしゅ）な計算が行われます。これを「ビット演算」といいます。

3.1. ビット演算の種類

先ほど、2つの色のデータを混ぜたときには、以下のようなルールでビット演算を行っていました。

・0と0を合わせた位は0になる。
・0と1を合わせた位は1になる。
・1と1を合わせた位は1になる。

2つのデータのうち片方だけでも1であれば、その位は1になる、という計算です。
この計算を「OR（オア）演算」といいます。

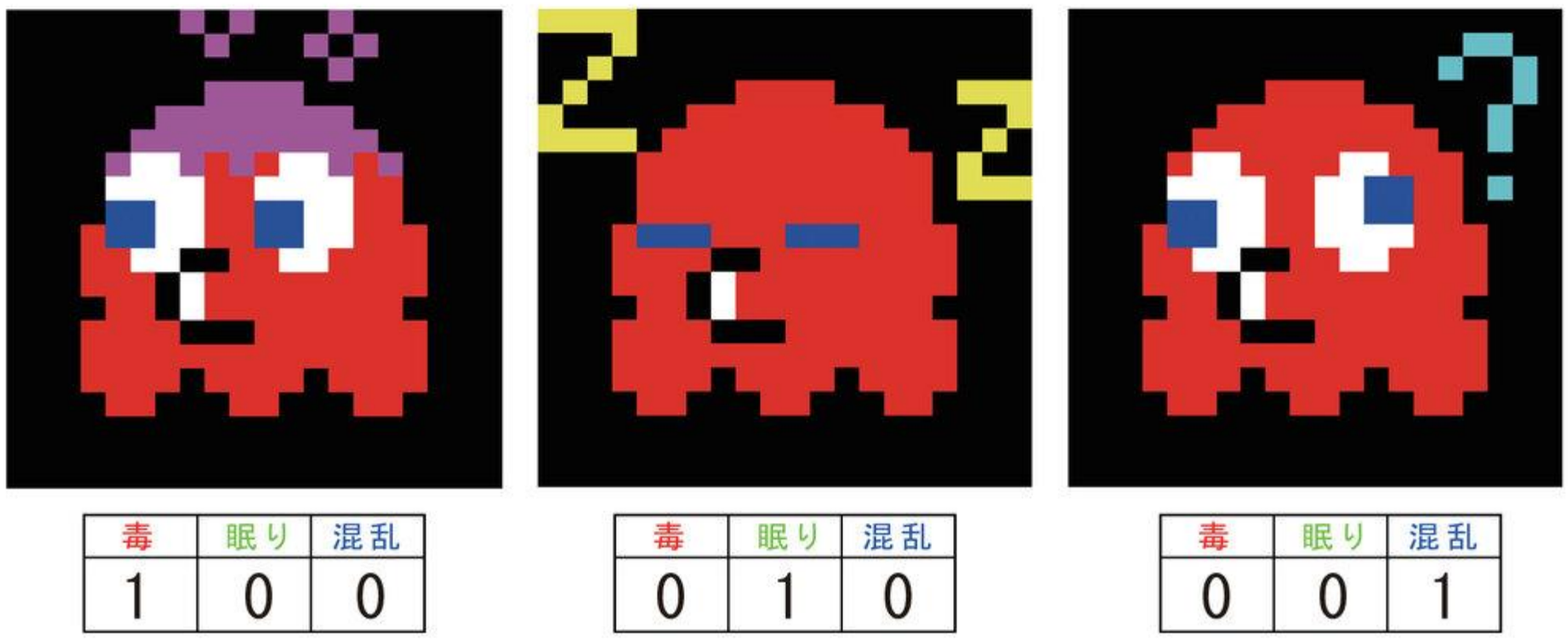
これとは別に、以下のような演算もあります。

「AND（アンド）演算」
・0と0を合わせた位は0になる。
・0と1を合わせた位は0になる。
・1と1を合わせた位は1になる。

エックスオア
「XOR演算」
・0と0を合わせた位は0になる。
・0と1を合わせた位は1になる。
・1と1を合わせた位は0になる。

「2つのデータを合わせる」だけなのに、ルールが何種類も存在するのです。なぜでしょうか？その理由を、ロールプレイング・ゲームのプログラマーになったつもりで考えてみましょう。

ロールプレイング・ゲームには、キャラクターが「毒」「眠（ねむ）り」「混乱（こんらん）」といった状態異常（じょうたいいじょう）になってしまうものが多いですよね。ためしにこの3つの状態異常（じょうたいいじょう）を、2進数のデータで管理してみましょう。



毒	眠り	混乱
1	0	0

毒	眠り	混乱
0	1	0

毒	眠り	混乱
0	0	1

図3-4　状態異常を各ビットで管理する

各状態異常になったとき、対応する位を「1」とする考え方です。
毒状態（100）のキャラクターに混乱攻撃（001）をすれば、キャラクターは「毒＋混乱」の状態（101）になります。
この場合、100 と 001 という 2 つのデータを OR 演算し、101 というデータにしています。

毒	眠り	混乱
1	0	0

＋

毒	眠り	混乱
0	0	1

＝

毒	眠り	混乱
1	0	1

図3-5　毒+混乱になった「たこ？」

やってみよう！

「毒＋混乱」の状態をつくるとき、2つのデータをAND演算してはだめなのかな？
ためしにAND演算をしてみて、どうなるか確認してみよう！

000

チャレンジ課題

毒状態だけを治療する「毒消し草」を考えてみよう。どのようなデータを、どの演算で合わせればよいかな？

ヒント

以下の条件で考えてみよう！
・毒状態の位は、治療前の値に関係なく「0」になるようにする。
・それ以外の状態の位は、治療前の値をそのまま維持する（毒以外の状態異常は治療しない）。

データ：　011　　演算法：　AND

場合によって、適切な演算法がいろいろあるということですね。
現代のゲームプログラミングでは、このようにデータを直接変更するようなプログラムをかくことは比較的少なくなりましたが、プログラムというのは「持っている知識をどう活用すればより効率的になるか」を考える機会が非常に多いです。こういった基礎的な知識は、持っておくといつか役立つかもしれませんよ。

豆知識

「OR」「AND」「XOR」はすべて、2つのデータを合わせる際の演算です。しかしビット演算にはほかに「1つのデータの値を変更する」ための演算もありますので、これも紹介します。

「シフト演算」
・指定されたケタ数だけ、「1」の場所を左または右に移動する。
　例：「00101000」を右に2ケタだけシフトさせると「00001010」になる。

「NOT演算」
・0だった位は1に、1だった位は0になる。
　例：「00101000」にNOT演算をすると「11010111」になる。

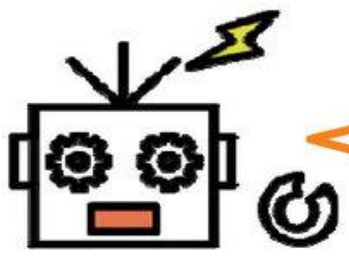
コラム　ビット演算子

プログラム内でビット演算を行いたい場合のかきかたを紹介します。
2つのデータを「A」「B」とするとき、以下のような記号を用いて数式にします。

OR演算	A | B
AND演算	A & B
XOR演算	A ^ B
シフト演算	A >> 1　(1ケタ右にシフトさせる)　　A << 3　(3ケタ左にシフトさせる)
NOT演算	~A

４．まとめ（目安５分）

今回は、コンピューターのメモリーや２進数・16進数、配列の使い方を勉強しました。コンピューターが理解できる0と1のデジタルの世界を知ることができたでしょうか。
次回はもう少し深く、データ型について勉強しましょう。メモリーの話で、世の中の製品は結構地道な作業で効率的に動かされているのだ、ということがわかってきたでしょうか。また、メモリーの演算を見ることで実はコンピューターの中身ってこんなものだったのだと理解できたと思います。キャラクターのかき方は、ちょっとゲームっぽいですよね。メモリーを使いこなしていくとキャラをつくれたり、不思議な動きをさせたりと色々とできるようになってきます。次回は「メモリーを使いこなす！」ということをして、ロボットがより賢く動かせるようなスキルを学んでいきます。それでは、また次回っ！



お部屋もプログラムも整理が大事なのね。

《次回必要なもの》

次回は、以下のパーツを持ってきてください。

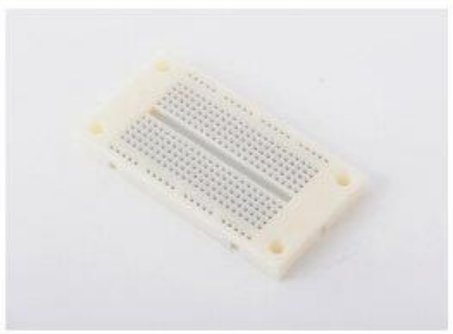
ラジオペンチ 1	USB ケーブル 1	マイコンボード 1	ロボプロシールド 1
			
301 ブレッドボード 1	ジャンパー線 65	抵抗 (100 Ω /1k Ω /10k Ω) 各10	タクトスイッチ 10
			
緑色 LED 10	姿勢検出シールド 1	液晶ディスプレイシールド 1	
			

図4-0　次回必要なもの

講

○以下の理解度を確認してください。
・メモリーとビット演算を学ぶ
・変数の宣言と種類を学ぶ
・液晶ディスプレイにキャラクターを表示させる
○次回、第4回は「キャラクターを動かそう」であることを告知してください。

P.16　チャレンジ課題①　解答例

```
#include <TFT.h>
#include <SPI.h>

TFT TFTscreen = TFT(A2, 1, 9);          //液晶ディスプレイを使うときのオマジナイ

const char chara[] = {
      1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
    , 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
    , 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
    , 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
    , 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0
    , 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0
    , 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0
    , 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0
    , 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0
    , 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0
    , 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0
    , 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0
    , 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0
    , 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
};

void setup(void){
    TFTscreen.begin();                  //液晶ディスプレイ表示
    TFTscreen.background(0, 0, 0);      //背景色は黒
    TFTscreen.stroke(255, 255, 255); //線の色は白
for(int j = 0; j < 14; j++){
    for(int i = 0; i < 14; i++){
        TFTscreen.stroke(255 * chara[14 * j + i], 255 * chara[14 * j +
i], 255 * chara[14 * j + i]);
        TFTscreen.point(i, j);
    }
}
}

void loop(){
}
```

P.16　チャレンジ課題②　解答例

```
#include <TFT.h>
#include <SPI.h>

TFT TFTscreen = TFT(A2, 1, 9);          //液晶ディスプレイを使うときのオマジナイ

const char chara[] = {
      0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0
    , 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0
    , 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0
    , 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0
    , 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0
    , 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0
    , 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1
    , 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1
    , 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
    , 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0
    , 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1
    , 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
    , 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1
    , 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0
};

void setup(void){
    TFTscreen.begin();                  //液晶ディスプレイ表示
    TFTscreen.background(0, 0, 0);      //背景色は黒
    TFTscreen.stroke(255, 255, 255); //線の色は白
for(int j = 0; j < 14; j++){
    for(int i = 0; i < 14; i++){
        TFTscreen.stroke(255 * chara[14 * j + i], 255 * chara[14 * j +
i], 255 * chara[14 * j + i]);
        TFTscreen.point(i, j);
    }
}
}

void loop(){
}
```