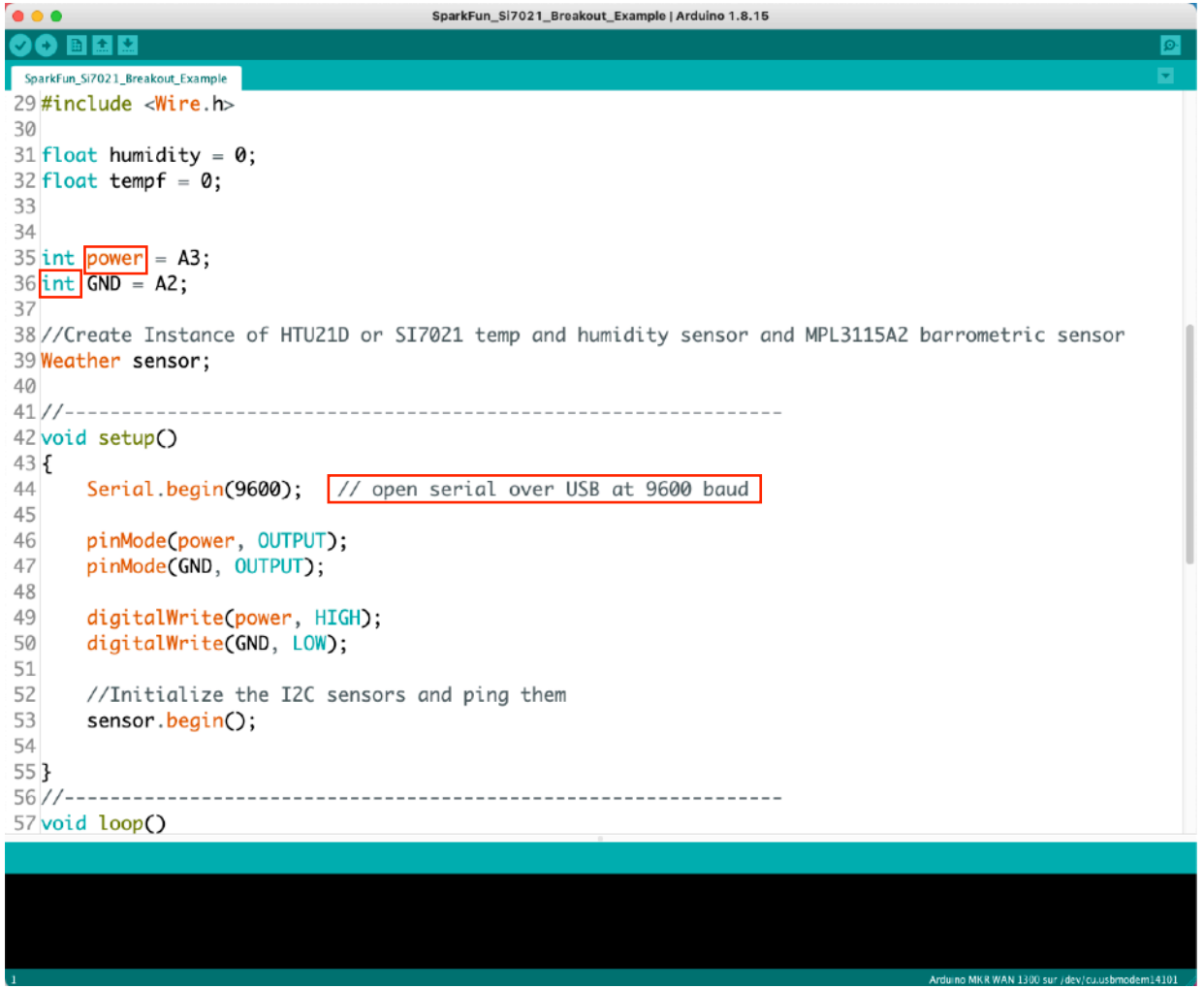
Bien programmer sous ARDUINO

SYNTAXE ET LES COULEURS

Des couleurs pour clarifier le statut des éléments :

- Orange : mots-clés reconnus par le langage Arduino
- Bleu : mots-clés reconnus par le langage Arduino (constantes, types, …)
- Vert : macro
- Gris : les commentaires
  - // sur 1 ligne tout caractère après est un commentaire
  - /* sur plusieurs lignes */



Penser comme un développeur

- Objectifs d'un *bon* développement, le programme doit
  - **s'exécuter**
  - **répondre aux besoins**
  - être **maintenable** ..... *compréhension facile, correction* des bugs et *mauvaises interprétations*
  - être **extensible** ..... *modifiable et évolutif*
  - être **réutilisable**

- Adoptez les bonnes pratiques
  - validées par des armées de développeurs
  - **Adoptez les plus basiques** au plus tôt (les mauvaises habitudes s'installent vite)
- Comment ?
  - Bien **écrire** son code..... *Style* et conventions
  - Bien **construire** son programme..... Quelques principes
  - Bien **travailler** avec le code..... ~~des outils pour mieux travailler~~

Bien écrire : *le Style*

- le code
  - **présentation** doit favoriser sa **compréhension** et permettre **une lecture rapide**,
  - s'adresse à des **humains** (la compilation au processeur), profitez-en !

  - sa mise en forme (**graphisme)** doit révéler une structure logique

  - Faciliter sa lecture
    - **Aérer** pour distinguer les parties du code
    - **Regrouper** les instructions pour focaliser l'attention
    - **Indenter** pour montrer les imbrications
    - **Commenter** pour expliquer ce qui doit l'être

le Style : Aérer pour distinguer

- Guider le regard
  - **L'interligne**, c'est la **ponctuation** qui donne le rythme la lecture

```
 9 void loop() {
10   //instructions entassées
11   b=digitalRead(p1);
12   int s=analogRead(p4);
13   byte c=digitalRead(p2);
14   long v=map(s,0,1024,-40,80);
15   if(v<0&&c==HIGH)digitalWrite(p3,LOW);
16   if(v>35&&c==LOW)digitalWrite(p3,HIGH);
17   if(b==HIGH){x=x+v;
18   Serial.print(x);}
19 }
```

Compilation terminée.

```
 9 void loop() {
10   // instructions aérées
11   b = digitalRead(p1);
12   int s = analogRead(p4);
13   byte c = digitalRead(p2);
14   long v = map(s, 0, 1024, -40, 80);
15
16   if(v < 0 && c == HIGH) digitalWrite(p3, LOW);
17   if(v > 35 && c == LOW) digitalWrite(p3, HIGH);
18
19   if(b == HIGH) {
20     x = x + v;
21     Serial.print(x);
22   }
```

Compilation terminée.

le Style : Regrouper pour focaliser

- Regrouper vos instructions
  - Par ilots logique, par nature, par étape, …
  - Pour que la compréhension soit « instinctive »
  - Sans que l'attention ne s'éparpille

```
 9 void loop() {
10   // instructions aérées
11   b = digitalRead(p1);
12   int s = analogRead(p4);
13   byte c = digitalRead(p2);
14   long v = map(s, 0, 1024, -40, 80);
15
16   if(v < 0 && c == HIGH) digitalWrite(p3, LOW);
17   if(v > 35 && c == LOW) digitalWrite(p3, HIGH);
18
19   if(b == HIGH) {
20     x = x + v;
21     Serial.print(x);
22   }
```

Compilation terminée.

```
 9 void loop() {
10   // instructions regroupées
11   int s = analogRead(p4);
12   long v = map(s, 0, 1024, -40, 80);  // C
13
14   byte c = digitalRead(p2);  // Lecture de c & utilisation
15   if(v < 0 && c == HIGH) digitalWrite(p3, LOW);
16   if(v > 35 && c == LOW) digitalWrite(p3, HIGH);
17
18   b = digitalRead(p1);  // Lecture de b & utilisation
19   if(b == HIGH) {
20     x = x + v;
21     Serial.print(x);
22   }
23 }
```

Compilation terminée.

Calcul de v

Lecture de c & utilisation

Lecture de b & utilisation

le Style : Indenter pour montrer

- Montrer les imbrications
  - Favoriser une lecture *verticale* qui permet d'ignorer des niveaux d'imbrications
  - Sur Arduino IDE ➔ CTRL+T
  - ATTENTION, **trop de niveaux** (> 3) , votre code devient difficile à maintenir ➔ **linéariser** !

```
 9 void loop() {
10   if ( a && b ) {
11     if (c && c > b ) {
12       b=1; v=a*c;
13     }else if (!c && a > b) {
14       a=1; v=b*c;
15     }}else a = 100;
16 }
```

Compilation terminée.

```
void loop() {
  if ( a && b ) {
    if ( c && c > b ) {
      b = 1;
      v = a * c;
    }
    else if ( !c && a > b ) {
      a = 1;
      v = b * c;
    }
  }
  else a = 100;
}
```

Si a && b == false…

…aller directement ici

```
void loop() {
  if ( a && b ) {
    if ( c && c > b ) {
      b = 1;
      v = a * c;
    }
    else if ( !c && a > b ) {
      a = 1;
      v = b * c;
    }
  }
  else a = 100;
}
```

le Style : Commenter pour expliquer

- Les commentaires vous donnent la parole
  - **Exprimez** vos idées, expliquez, éclaircissez
- Commentaires de documentation
  - Pour expliquer l'idée générale du programme aux autres
  - Pour éclaircir votre vision du programme
- Commentaires en ligne
  - A côté des déclarations
  - Dans les passages tortueux
  - Pour marquez la fin des blocs
- N'oubliez pas de les mettre à jour

```
/**
  Sun Tracker
  Le tracker fonctionne avec 4 photoresistances
  disposées en croix de part et d'autre de séparations.

  Le programme calcule le différentiel de luminosité et
  en déduit la direction de plus forte intensité.

  Ls contrôle du tracker se fait grâce à 3 bouttons
  B1, B2 et B3.

  Calibrer le tracker; appui long de 3s sur B1
  Tester les servos moteur; appui long de 3s sur B2
  ...

 */
```

```
float  v; // vitesse du robot (speed)

// Si la distance théorique et la distance
// calculée sont différentes de 0.1 m,
// alors relancer recalibrage
if ( abs(d - v*t) > 0.1 ) {

    } // fin recalibrage capteur
  } // fin boucle vérification système
} // fin initialisation
```

Bien écrire : **conventions**

Les conventions **limitent les incompréhensions** et **donnent du sens** à vos blocs d'instructions

- Quelques conventions simples
  - Convention de langue
  - Convention de nommage
    - sur les noms de variables
    - sur les noms de fonctions

conventions de langue

Quelle langue utiliser ?

- English is better
  - **Esthétique** → meilleure intégration avec Arduino
  - **Universalité** → Contexte de recherche internationale
    - *WTF is this ''`activerCompteARebourdPourMecanismeDeSabordage()`''*
      => countdownForScuttlingMechanismEnabled
  - **Plus précise** →Distinction verbe/nom/forme passée quasi impossible en français sans accent
    - « `chienMange` » **vs** « `dogEating` » **vs** « `eatenDog` »
- Convention mixte
  - Si grande subtilité dans le modèle et/ou dialogue avec acteurs anglophobes
  - Nom de variable en français, verbe en anglais
    - `readAzimutCompasGyroscopiqueAnalogique()`

conventions de nommage

- Les noms doivent être **significatifs**
  - Ne soyez pas avare sur les lettres (min 3),
  - Un nom court ou sans signification (i, j, k) ➔ variable non stratégique

- Construction des noms
  - camelCase → **`avecDesBossesCommeLeDromadaire`**
    - arduino, java, php, c#, …
  - snake_case → **`avec_des_traits_comme_des_serpents`**
    - python, perl, ..
  - Question de goût, mais restez homogène

```
int valeurPressionSondeExterieure;
int externalPressureSensorValue;
```

```
int external_pressure_sensor_value;
int mesure_pression_sonde_exterieure;
```

- Les constantes toujours en majuscule
  - et donc snake_case
  - tout le reste en non-majsucule ☺

```
const int NB_DE_ROUES = 4;
```

conventions de nommage

Variables sous arduino

- emplacement réservé dans la mémoire du microcontrôleur
- la taille est directement lié au type
- identifiée par un nom qui est associé à une valeur
  nom = adresse qui permet un accès direct à la valeur

Il est nécessaire de déclarer les variables pour leurs réserver un espace mémoire adéquat (1 à 8 octets).
On déclare une variable en spécifiant son type, son nom (avec la possibilité de lui assigner une valeur initiale).

```
int variable;
int variable = 45;  // int : type,  variable : nom, 45 : initialisation

const int PIN_LED = 10;   // constante

volatile int state = LOW; // volatile : !optimisation compilateur
```

Les différents types de variables sont :

- void // utilisé dans les fonctions
- boolean // 8 bits : true ou false
- char // 8 bits : 'a', -128 à 127
- unsigned char, byte // 8 bits : 0 à 255
- int // 16 bits : -32768 à 32767
  // 32 bits Due : -2147483648 à 2,147,483,647
- unsigned int, word // 16 bits : 0 à 65535
  // 32 bits Due : 0 à 4294967295
- long // 32 bits : -2147483648 à 2,147,483,647
- unsigned long // 32 bits : 0 à 4294967295
- short // 16 bits : -32768 à 32767
- float // 32 bits : 3.4028235E+38 à -3.4028235E+38
- double // 32 bits : 3.4028235E+38 à -3.4028235E+38
  // 64 bits de précision sur la Due
- string // char array (C)
- String // object (C++)

conventions de nommage

Exemples de déclaration de variables

- Utiliser les noms des éléments (capteurs, actionneurs) du projet comme noms des variables du programme permet de clarifier le rôle des différentes variables.

- Exemple, pour choisir une variable qui a pour rôle de définir la broche sur laquelle est connectée une LED, nous conseillons de nommer la variable comme suit :

  - int brocheLed = 13;
  - int broche_led = 13;
  - int brocheDeLaLed = 13;
  - int broche_de_la_led = 13;
  - int **pinLed** = 13;
  - int **pin_led** = 13;
  - int pinDeLaLed = 13;
  - int pin_de_la_led = 13;

  - int iBrocheLed = 13;
  - int i_broche_led = 13;
  - int iBrocheDeLaLed = 13;
  - int i_broche_de_la_led = 13;
  - int **iPinLed** = 13;
  - int **i_pin_led** = 13;
  - int iPinDeLaLed = 13;
  - int i_pin_de_la_led = 13;

conventions : état et action

- Les *variables* sont des *noms* exprimant un *état*
  - `nbCanaux` , `tensionSortie`, `porteEstFermee`
- Les *fonctions* sont des *verbes* exprimant une *action*
  - Les **interrogations** ou **lecture** d'un état commencent par les formes *conjuguées* **être** et **avoir**
    - `estPorteOuverte()`, `aRecuMessage()`, `isDoorOpen()`, `hasReceivedMessage()`
  - Les **autres** actions (calcul, modification) indiquent par l'usage de **l'infinitif** ce qu'elles font
    - `calculerPointDeRosee(float t)`, `ouvrirPorte()`, `opendDoor()`
  - Les actions sont **toujours** des fonctions avec des parenthèses à la fin

- Montrez la *finalité* de l'action
  - Le **quoi**, pas le **comment**
  - Cacher ce qui peut nuire à la compréhension du code
    - La mise en œuvre, les étapes intermédiaires, …

conventions : état et action

Les fonctions

- c'est un bloc d'instructions que l'on peut appeler plusieurs fois, n'importe où dans le programme
- Arduino possède déjà un certain nombre de fonctions : analogRead(), digitalWrite() ou delay()…
- Il est possible de déclarer ses propres fonctions :

```
type nomFonction () {
} // sans arguments

type nomFonction (type arg1, …) {
} // avec arguments
```

Pour déclarer la fonction clignote ci-dessous, placez-vous en dehors d'une fonction (colonne 0) et écrivez ceci :

```
void clignote(int pin, int tps_ms) {
  digitalWrite(pin, HIGH);
  delay(tps_ms);
  digitalWrite(pin, LOW);
  delay(tps_ms);
}
```

Pour exécuter une fonction, il suffit d'écrire son nom avec les parenthèses (et ses arguments si elle en a) :

```
// la led builtin clignote toutes les 2 secondes
clignote(LED_BUILTIN, 1000);
```

conventions : variables significatives

- Le nom de la variable doit être **sans ambiguïté**
- Un seul coup d'œil permet de comprendre de quoi on parle

```
int s = analogRead(p4);
long v = map(s, 0, 1024, -40, 80);

byte c = digitalRead(p2);
if( v < 0 && c == HIGH ) digitalWrite(p3, LOW);
```

```
int signalCapteurExt = analogRead(pinCapteurExt);
long tempExterieure = map(signalCapteurExt, 0, 1024, -40, 80);

byte etatPorteNord = digitalRead(pinContactPorteNord);
if( tempExterieure < 0 && etatPorteNord == HIGH ) {
    digitalWrite(pinMoteurPorteNord, LOW);
}
```

conventions : actions significatives

```
int signalCapteurExt = analogRead(pinCapteurExt);
long tempExterieure = map(signalCapteurExt, 0, 1024, -40, 80);

byte etatPorteNord = digitalRead(pinContactPorteNord);
if(tempExterieure < 0 && etatPorteNord == HIGH) {
    digitalWrite(pinMoteurPorteNord, LOW);
}
```

```
int signalCapteurExt = lireCapteurTemperature(pinCapteurExt);
long tempExterieure = map(signalCapteurExt, 0, 1024, -40, 80);

byte etatPorteNord = estPorteNordOuverte();
if(tempExterieure < 0 && etatPorteNord == HIGH) {
    fermerPorteNord();
}
```

- Cacher la **mise en œuvre** quand elle n'est nécessaire à la compréhension de la logique

conventions : se concentrer sur l'essentiel

- Réduisez la dispersion et cachez les étapes intermédiaires

```
int signalCapteurExt = lireCapteurTemperature(pinCapteurExt);
long tempExterieure = map(signalCapteurExt, 0, 1024, -40, 80);
```

```
byte etatPorteNord = estPorteNordOuverte();
if( tempExterieure < 0 && etatPorteNord == HIGH ) {
    fermerPorteNord( );
}
```

```
long tempExterieure = lireTemperatureExt();
```

```
byte etatPorteNord = estPorteNordOuverte();
if(tempExterieure < 0 && estPorteNordOuverte()) {
    fermerPorteNord();
}
```

- Sur quelques lignes, c'est insignifiant;
- Sur plusieurs pages, c'est fondamental

Bien construire son programme

- Le « tombe-en-marche » est votre pire ennemi
  - Ca marche, mais vous ne savez pas pourquoi, ce n'est plus de la programmation, c'est de la magie.
- Décomposer votre code en petits blocs
  - Une fonction qui s'étale sur quelques hauteurs d'écrans est difficile à maintenir
  - Fabriquez vos briques de Lego ☺
- Quelques principes de décomposition
  - Principe de **lisibilité** (déjà vue)
  - Principe de **responsabilité**
  - Principe de **délégation anticipée**
  - Principe de **factorisation**
  - **Linéarisation** des blocs conditionnels
  - Et encore bien d'autres https://en.wikipedia.org/wiki/List_of_software_development_philosophies

Principe de responsabilité
ou de délégation

- Responsabilité ?
  - **Action** ou **ensemble d'actions**, *dont les détails* peuvent être **délégués** à **un bloc de code identifié** tels que une ***fonction***, une ***macro***, une ***classe***, une ***bibliothèque***
  - Une *responsabilité* peut regrouper des *sous-responsabilités*

- Pourquoi les rechercher ?
  - **Testabilité** → Il est plus facile de tester et s'assurer du bon fonctionnement d'un bloc bien défini et délimité
  - **Découplage** → pouvoir modifier un calcul, une mise en œuvre sans tout remettre en cause
  - **Modularité** → réutilisation possible

Exemples de responsabilités

- ‘’Obtenir la température extérieure”
  - fonction `readExternalTemperature()`
  - regrouper ‘’*lecture du capteur*” et ‘’*conversion*” pour ne fournir que ce qui nous intéresse: la température
- ‘’Gérer communication port série”
  - classe `HardwareSerial`
  - ~ 500 lignes de code cachées
  - Une trentaine de méthodes

```
Serial.begin(9600);
Serial.print(" Hello World !");
```

- ‘’Actionner actuateur (en toute sécurité)”
  - méthode `Actuator::goToPosition(int x, int y)`
  - ‘’*Vérifier compatibilité avec limites d’usage*”,
  - ‘’*lancer initialisation au besoin*”,
  - ‘’*mettre à jour variables d’état*”

Principe de délégation anticipée

- Remettre à plus tard ce qu'on ne sait pas faire maintenant
  - **Détecter** une responsabilité
    - **Identifier** ses entrées et ses sorties
    - **Ecrire** la fonction/classe/interface/macro/bibliothèque *à minima*
    - **Falsifier** son fonctionnement au besoin
  - Et **avancer …**
- Objectif : Construire le programme
  - **Ne pas se perdre** en chemin: ‘’le diable se cache dans les détails’’
  - Obtenir **l’idée générale** du fonctionnement, ‘’sentir’’ le programme
  - Base de discussion avec les collègues ou soi-même
- Mettre un garde fou
  - Ne pas laisser l’illusion persister que cela fonctionne !
  - Avec des macros ou des commentaires « TODO »

```
void loop( ) {
   if ( mode == EMERGENCY ) {
      sendSOS( );
   }
  // …
}

void sendSOS( ) {
   // TODO
   Serial.print(" SOS working progress…" );
}Serial.begin(9600);
```

```
#define MODE_DEV 1
#define sayOoops(x) Serial.print(x)

void sendSOS( ) {
   #ifndef MODE_DEV
   sayOoops(" sendSOS…")
   #endif
}
```

Principe de factorisation

- Don’t repeat yourself (DRY)
  - La duplication de code est source d’erreur
  - Les répétitions doivent être **identifiées**, **isolées** et **placées** dans une fonction/macro/classe/bibliothèque
  - Limitez les définitions en **un et un seul** emplacement

- Code récurrent de projet en projet
  - Ne recopiez pas le code de programme en programme,
  - Externalisez le dans une bibliothèque

```
void loop( ) {
    // …
    if (a && b && (c || d )) {
        doIt( );
    }
    if (e && f && (g || h )) {
        doIt( );
    }
    if (i && j && (k || l )) {
        doIt( );
    }
    // …
}

/* explication de la responsabilité */
void doItWhenCondition(byte a,byte b,byte c,byte d) {
    if ( a && b && (c || d )) {
        doIt( );
    }
}

void loop ( ) {
    // …
    doItWhenCondition(a, b, c, d);
    // …
    doItWhenCondition(e, f, g, h);
    // …
    doItWhenCondition(i, j, k, l);
```

Linéarisation des conditions

- Cassez la structure pour limiter les imbrications
  - Posez les conditions et alignez les en pensant à l'action souhaitée
    - travaillez en mode « Agir QUAND conditions »
  - Modification des conditions aisées

```
void doSomething() {
    while( !isCarAway() ) {
        if( isCarOpened() ) {
            if (isEnginOn()) ) {
                if ( areAllPassengerIn() ) {
                    if ( isDriverMajor() ) {
                        goOn();
                    } else {
                        goToPrison();
                        break;
                    }
                } else if ( isTherePassengerSomewhere() ) {
                    waitForPassengers();
                }
            } else {
                startEngine();
            }
        } else {
            openTheCar();
        }
    }
}
```

4 niveaux max

1 niveau

```
void doSomething() {

    while( ! isCarAway() ) {

        byte carOpened      =  isCarOpened();
        byte enginOn        =  isEnginOn();
        byte passengersIn   =  areAllPassengerIn();
        byte driverMajor    =  isDriverMajor();
        byte passengersSomewhere =  isTherePassengerSomewhere();

        if ( ! carOpened ) {
            openTheCar();
        }
        if ( carOpened && !enginOn ) {
            startEngine();
        }
        if ( carOpened && enginOn && !passengersIn && passengersSomewhere ) {
            waitForPassengers();
        }
        if ( carOpened && enginOn && passengersIn && driverMajor ) {
            goOn();
        }
        if ( carOpened && enginOn && passengersIn && !driverMajor) {
            goToPrison();
            break;
        }
    }
}
```

Séparer logique et mise en oeuvre

```
long tempExterieure = lireTemperatureExt();
if(tempExterieure < 0 && estPorteNordOuverte()) {
  fermerPorteNord();
}
```

Fonction appelée
à l'exécution du programme

```
// par fonction
byte estPorteNordOuverte() {
    return digitalRead(pinContactPorteNord) == HIGH;
}
```

Ou

Pré-processeur
Substitution de texte
avant la compilation
du programme

```
// par pre-processeur
#define estPorteNordOuverte() (digitalRead(pinContactPorteNord)==HIGH)
```

- syntaxes générales pour déclarer une macro :
  - #define <IDENTIFIANT> <contenu de remplacement> // type objet
  - #define <IDENTIFIANT>(<liste des paramètres>) <contenu de remplacement> // type fonction avec paramètres

- par convention les noms des macros sont écrites en majuscules, exemples :

```
#define   PI   (float)3.14159
#define   MAX(a,b)    a > b ? a : b
#define   PIN_LED    13     // remplace const int PIN_LED = 13
```

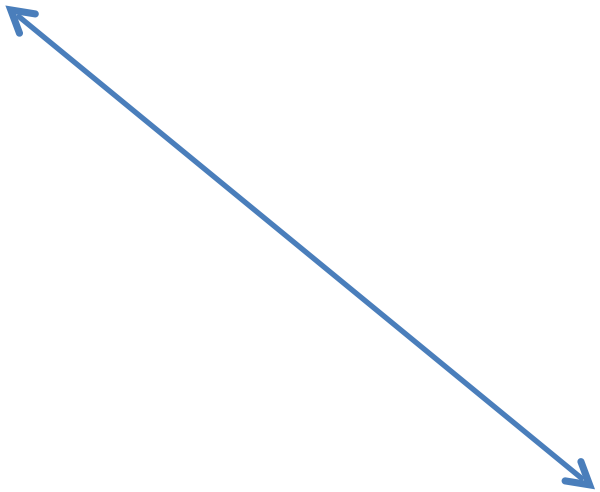
Séparer logique et mise en oeuvre

Configuration matérielle

Exploitation bas-niveau

Niveau modélisation

```
#define pinContactPorteNord      2
#define lireContactPorteNord()   digitalRead(pinContactPorteNord)
#define estPorteNordOuverte()    (lireContactPorteNord()==HIGH)
```



```
long temperatureExterieure = lireTemperatureExt();
if ( temperatureExterieure < 0 && estPorteNordOuverte() ) {
  fermerPorteNord();
}
```

Séparer logique et mise en oeuvre

```
#define pinContactPorteNord      2
#define lireContactPorteNord()   digitalRead(pinContactPorteNord)
#define estPorteNordOuverte()    (lireContactPorteNord()==HIGH)
```

Changement hardware ou accès

Autre configuration matérielle

Autre exploitation bas-niveau

Même modélisation

```
#define bitContactPorteNord      5
#define lireContactPorteNord()   PORTD |= (1 << bitContactPorteNord )
#define estPorteNordOuverte()    (lireContactPorteNord()==HIGH)
```

```
long temperatureExterieure = lireTemperatureExt();
if ( temperatureExterieure < 0 && estPorteNordOuverte() ) {
  fermerPorteNord();
}
```

L'utilisation de la méthode ne **CHANGE PAS** dans le code principal

S’améliorer

- Quelques écueils à éviter
  - Tout garder en soi: exprimer le problème aide à le résoudre
    - Verbaliser, dessiner, parler à votre crayon
  - Ne pas attendre le bout du tunnel
    - Assurez-vous que cela compile régulièrement
  - Ne pas réinventer pas la roue, s’inspirer des codes et librairies des développeurs confirmés,
    - Stackoverflow, blogs, forum
  - Ne **jamais copier** un bout de code sans chercher à **comprendre** (un peu)
  - « Optimization Is Evil »
    - Ne pas chercher à optimiser la performance de suite
- You Ain't Gonna need it (YAGNI)
  - S’avoir s’arrêter et ne pas ajouter de fonctionnalités qui ne sont pas demandées
- Read The F*cking Manual (RTFM)
  - Prendre le temps de comprendre en fait gagner

Structures de contrôle

Les structures de contrôle sont des blocs d'instructions qui s'exécutent suivant des conditions.

Il existe une dizaine de types de structure qui peuvent être regroupées en 4 :

- **if**
  if...else
- for
- switch case
- while
  do... while

exécute un code si certaines conditions sont remplies
et éventuellement exécutera un autre code avec sinon.

exemple :

```
// si la valeur du capteur dépasse le seuil
    if ( valeur_capteur > seuil ) {
    // appel de la fonction

    }
    else {
    // TODO
    }
```

Structures de contrôle

Les structures de contrôle sont des blocs d'instructions qui s'exécutent suivant des conditions.

Il existe une dizaine de types de structure qui peuvent être regroupées en 4 :

- if
  if...else
- **for**
- switch case
- while
  do... while

exécute un code un nombre de fois prédéfini (évite les boucles infinies)

exemple :

```
// pour i de 0 à 255, par pas de 1
    for ( int i=0 ; i<=255 ; i++ ) {
        analogWrite(PIN_PWM, i);
        delay(10);
    }
```

Structures de contrôle

Les structures de contrôle sont des blocs d'instructions qui s'exécutent suivant des conditions.

Il existe une dizaine de types de structure qui peuvent être regroupées en 4 :

- if
  if...else
- for
- **switch case**
- while
  do... while

faire un choix entre plusieurs codes parmi une liste de possibilités prédéfinis

exemple :

```
// faire un choix parmi plusieurs messages reçus
    switch (message) {
        case 0 : //cas du message 0
            digitalWrite(3, HIGH);
            digitalWrite(4, LOW);
            digitalWrite(5, LOW);
        break;
        case 1 : // cas du message 1
            digitalWrite(3, LOW);
            digitalWrite(4, HIGH);
            digitalWrite(5, LOW );
        break;
        case 2 : // cas du message 2
            digitalWrite(3, LOW);
            digitalWrite(4, LOW);
            digitalWrite(5, HIGH);
        break;
        default:
        break;
    }
```

Structures de contrôle

Les structures de contrôle sont des blocs d'instructions qui s'exécutent suivant des conditions.

Il existe une dizaine de types de structure qui peuvent être regroupées en 4 :

- if
  if...else
- for
- switch case
- **while**
  do... while

exécute un code tant que certaines conditions sont remplies
exemple (code non optimisé) :

```
// tant que la valeur du capteur est supérieure à 250
      while ( valeur_capteur > 250 ) {
      // on active la sortie 5
           digitalWrite(5, HIGH);
      // on envoi le message "1" au port série
           Serial.println(1);
      // boucle tant que valeur_capteur est supérieure à 250
      }
     Serial.println(0);  // on envoi le message "0" au port série
     digitalWrite(5, LOW);  // on désactive la sortie 5
```

Pour Résumer

Un langage de programmation passe par l'apprentissage d'un vocabulaire et d'une syntaxe précise.

Nous avons vu quelques unes de ces règles ainsi que d'autres éléments à prendre en considération dans l'écriture d'un programme Arduino.

Si votre code nécessite un suivi (lecture, modification), c'est de votre responsabilité d'apporter une attention particulière à sa structure et d'y apposer des commentaires expliquant la logique que vous avez choisi.

Un code clair offre une meilleure lecture pour :

- modifier les configurations,
- localiser une source d'une erreur,
- le faire évoluer.

Pour Résumer

CONVENTIONS DE NOMMAGE

La syntaxe d'un langage de programmation se base généralement sur une « convention de nommage ».
Il est important de respecter cette nomenclature pour rester dans l'esthétique du code.

Exemple, dans la déclaration des variables, il faut écrire les constantes en majuscule :

```
const int CAPTEUR = A0; // CAPTEUR est défini comme une constante.
```

Pour les autres variables (sans const) et les fonctions il faut utiliser les noms en minuscule.

En cas de variables avec des noms composés, le premier mot est en minuscule et les autres commencent soit :

- par une majuscule, exemple : `int cetteAnf = 2022;`
- par un underscore, exemple : `int cette_anf = 2022;`

Pour les noms des fonctions, on applique la même règle, exemples :

- afficherValeurCapteur ();
- afficher_valeur_capteur ();

Pour Résumer

COMPACTER SON CODE A L'AIDE DE FONCTIONS

- si votre code commence à tenir une place importante
- si vous utilisez à plusieurs reprises les mêmes blocs d'instructions, (organiser et alléger son programme)

Les fonctions peuvent être écrites avant ou après la boucle principale

VERIFIER SON CODE AU FUR ET A MESURE

- compiler régulièrement son code avec le bouton « Verify » pour trouver des erreurs subtiles

Les erreurs apparaissent en bas de la fenêtre, dans la console.

FONCTIONS AUTOMATIQUES DANS L'EDITEUR

la fonction « auto-format » indente automatiquement votre code (se base sur les conventions de programmation pour formater).

Aller dans **Tools > Auto Format** (Outils > Formatage automatique) ou utiliser le raccourci correspondant.

http://fr.wikipedia.org/wiki/Convention_de_nommage

Pour Résumer

BIBLIOTHEQUES ARDUINO

- fonctions utilitaires, regroupées et mises à disposition des utilisateurs afin de ne pas avoir à réécrire des programmes (complexes)
- fonctions regroupées suivant un même domaine conceptuel (mathématique, graphique, tris, etc)

Arduino :

- plusieurs bibliothèques externes par défaut
- importer en sélectionnant > **Import Library** (Croquis > Importer une bibliothèque)

L'instruction suivante est alors ajoutée au début de votre programme :

#include <le_nom_de_la_bibliothèque.h> *// inclut au code le contenu de la bibliothèque*

Les fonctions qu'elle contient peuvent alors être appelées au même titre que les fonctions de base.

Une bibliothèque logicielle se distingue d'un exécutable par le fait qu'elle ne s'exécute pas "seule" mais est conçue pour être appelées par un autre programme.

Pour Résumer

BIBLIOTHEQUES PAR DEFAUT

- EEPROM : lecture et écriture de données dans la mémoire permanente.
- Ethernet : pour se connecter à Internet en utilisant le Shield Ethernet.
- Arduino Firmata : pour rendre l'Arduino directement accessible à des applications en utilisant un protocole sériel.
- LiquidCrystal : pour contrôler les afficheurs à cristaux liquides (LCD).
- SD : pour la lecture et l'écriture de données sur des cartes SD.
- Servo : pour contrôler les servomoteurs.
- SPI : pour communiquer avec les appareils qui utilisent le protocole de communication SPI (Serial Peripheral Interface).
- SoftwareSerial : pour établir une communication sérielle supplémentaire sur des entrées et sorties numériques (la carte Arduino dispose d'un seul port sériel hardware).
- Stepper : pour commander des moteurs « pas à pas ».
- Wire : pour interfacer plusieurs modules électroniques sur un bus de données utilisant le protocole de communication TWI/I2C.

Pour Résumer

BIBLIOTHEQUES PROVENANT DE TIERS

D'autres librairies sont disponibles en téléchargement à l'adresse suivante.

https://www.arduino.cc/reference/en/libraries/

Procédure d'installation :

- décompresser le fichier téléchargé
- enregistrer dans un répertoire appelé « **libraries** » situé dans :
  - Sur Linux et Windows, le dossier « **sketchbook** » qui est créé au premier lancement de l'application Arduino dans votre dossier personnel.
  - Sur Mac OS X, le dossier « **Arduino** » qui est situé dans le répertoire « **Documents** ».

Par exemple, pour installer la librairie **DateTime**, le fichier devra être déposé dans le dossier :

/libraries/DateTime de votre sketch (Croquis).

Pour Résumer

LE VIRTUAL PORTCOM / Moniteur série / Traceur série : L'objet Serial

```
int fct (int value) {
    Serial.print (" fct ");  // affiche sans retour ligne
    Serial.println (value, DEC);  // affiche en décimal et avec retour ligne
    return value;
}

void setup() {
    Serial.begin (9600);  // init Serial baudrate
}

void loop() {
    delay(1000);
    int x = fct(0) & fct(1);
}
```

Pour Résumer

PLUS D'INFORMATIONS SUR…

La liste exhaustive des éléments de la syntaxe du langage Arduino est consultable sur le lien suivant :

http://arduino.cc/fr/Main/Reference

Vous pouvez consulter des informations et des explications générales sur le langage de programmation sur le lien suivant :

http://fr.wikipedia.org/wiki/Langage_de_programmation