www.didel.com/educ/EduC-Clair.pdf

Commencer le C avec *Clair*

Vous savez taper sur un clavier, sauver vos fichiers ? Le copier-coller vous fait gagner du temps ? Vous voulez voir comment on programme sans vous noyer dans des tas de choses à mémoriser ? L'anglais ne vous est pas très familier, charger des programmes ça va tout juste, mais vous aimeriez bien savoir comment on agit sur un écran, ce qu'est un algorithme, le temps réel, etc.
Alors, même si vous êtes très jeunes (cela vous préparera à maîtriser les applications auxquelles vous pensez) ou plus très jeune (et découragés par la complexité actuelle de l'informatique), vous aurez du plaisir avec notre approche.

Note pour des enseignants/encadrants: Le but de ce document est de présenter un concept pédagogique et de faciliter son test dans quelques situations. La quantité d'explications et d'exercices est minimale. Si le concept est bon, un document avec un graphisme attrayant, influencé par des enseignants expérimentés, sera l'étape suivante.

Introduction

Clair permet d'apprendre à programmer en C en utilisant au début des notations en français avec quelques fonctions supplémentaires qui facilitent la compréhension et l'écriture de programmes simples. Arduino en français, cela existe http://www.duinoam.com/ mais c'est une voie qui isole et n'allège pas l'effort conceptuel et ne diminue pas tout ce qu'il faut mémoriser.
Passez quelques heures avec ***Clair*** et vous aurez vu si la programmation vous intéresse et comment continuer à apprendre à programmer.

L'environnement de programmation est le Edu-C compatible Arduino. Si tout ce que l'on peut faire avec Arduino vous attire, c'est une "mise en bouche" parfaite. Arduino encourage des kits pleins de composants à câbler pour un prix dérisoire. C'est bien, on apprend à mettre des fils dans des trous et à charger des programmes dont on explique chaque ligne, mais n'apprend pas à structurer.
Nous voulons ici interagir avec un montage simple et fiable et comprendre l'essence de la programmation. Une variante de cette documentation utilise le LCbot utilisé dans le MOOC EPFL, que vous pourrez continuer à utiliser si vous vous êtes intéressés à la programmation, voir www.didel.com/lc/Clair.pdf . Le Edu-C vous conduira plutôt vers une activité ludique.
Dans les deux cas, vous pourrez continuer avec ce qui est offert sur le Web et faire tout ce qui est à la mode et correspond à vos intérêts.

Librairie Clair.h

Arduino a ajouté les fonctions digitalWrite et quelques autres pour éviter de comprendre les manipulations de bits, et permettre un peu de portabilité au prix d'un ralentissement peu gênant pour les applications simples. Les nombreuses librairies Arduino permettent d'ajouter d'autres fonctions, en général liées à des périphériques.
Avec ***Clair*** on oublie le câblage. On travaille avec la librairie Clair.h qui connaît le câblage de la carte. L'utilisateur pense **fonction**, par exemple *"allumer la Led à gauche*", noté **`LedGOn;`** , et non pas comme avec Arduino **câblage**: *"activer la pin 5 avec l'état LOW qui allume"* , ce qui oblige à écrire chaque fois **`digitalWrite(5,LOW);`**

Les définitions des entrées-sorties sont complétées par des équivalents en français des instructions C: SI() RepeterSi() Iterer(). On se concentre sur les fonctions du C dans leur implémentation la plus simple, déjà très riche en exemples formateurs. Passer au C en utilisant `if() while() for(;;),` quand on a bien compris comment programmer les structures de base ne pose pas de problème et donne une compréhension plus profonde.

Expérience avec Edu-C

Arduino doit être installé. Cela a été fait de millions de fois, mais pour celui qui n'a pas l'expérience d'installer une appli ou un driver sur PC ou Mac, c'est souvent délicat. Pour nos explications, voir www.didel.com/ArduinoInstall.pdf
La librairie Clear.h est incluse avec chaque programme. Clear.h est un extrait des librairies de base EduC-h et Oled.h de la documentation complète de l'Edu-C. La transition vers Fun ou Educ (7 modules) sera naturelle.

Pour éviter le temps perdu à taper des programmes, tous les programmes de ce cours sont dans le fichier www.didel.com/educ/EduC-Clair.zip . Il faut le dézipper et créer un dossier facilement accessible qui contient des dossiers et dans chaque dossier un programme .ino et sa librairie Clair.h.
On clique sur le fichier.ino choisi, ce qui charge Arduino et le programme. Pour sauver une variante, on "Enregistre sous" en créant le nom d'un nouveau dossier qui va contenir le .ino.
Avant d'exécuter le premier programme, vérifiez les options dans "Outils".

1 Instruction Si

Chaque programme doit importer la librairie et l'initialiser. Arduino demande de mettre les instructions entre **`void loop(){`** et une accolade fermée **`}`** après la dernière instruction. Avant ce `loop`, il faudra déclarer les variables, s'il y en a.

Notre premier programme lit le poussoir et copie son état. Mais que veut dire copier, qu'est-ce qu'un état?
Le poussoir, c'est comme une variable, mais avec 2 valeurs, 0 et 1, faux et vrai, relâché et pressé.
Pour les Leds, c'est différent, on agit, le microcontrôleur active une sortie. Fonctionnellement, 1 pour allumer, 0 pour éteindre, mais électriquement, cela dépend du câblage.
On a sur le Edu-C deux poussoirs et deux Leds supérieures, avec des inscriptions PG PD LG LD
On a défini les états **`PousG`** (si pressé) **`!PousG`** (si relâché), **`PousD`** et **`!PousD`**.
Le signe ! signifie inverse en C.
Pour les Leds: **`LedGOn;`**, **`LedGOff;`**, **`LedGToggle;`**, **`LedDOn;`**,**`LedDOff;`**, **`LedDToggle`** (Toggle inverse).

Le premier programme "copie" le poussoir sur la Led. Si c'est pressé, on allume; si c'est relâché, on éteint. En C, il faut écrire deux lignes, entre les accolades de la boucle d'exécution loop()

```
// C11Copie.ino
#include Clair.h
void setup {SetupClair(); }
void loop() {
     Si(PousG)   {LedGOn;}
     Si(!PousG)  {LedGOff;}
 }
```

Il faut s'habituer aux règles du C: un point-virgule à la fin de chaque instruction, des parenthèses pour les conditions et les paramètres (même inexistants), des accolades pour regrouper les actions. Pour les espaces et les sauts de ligne on est libre et il faut soigner la lisibilité.

Que fait le processeur ? Il teste la ligne poussoir, exécute si vrai et passe plus loin. A raison de plusieurs instructions simples par microseconde, le poussoir est copié et s'il y a des rebonds de contact, chaque rebond est copié, mais on ne les voit pas à l'oeil (il faut un bon oscilloscope).

Exercice.
Compléter le programme C11Copie.ino pour aussi copier PousD sur LedD.
C'est aussi un bon exercice pour apprendre à faire du copier-coller, et à se concentrer sur les modifications.

Autrement

L'instruction Autrement {} suit immédiatement un Si(){} et dit ce qu'il faut faire si la condition est fausse.
Par exemple, si on presse, on allume L1, autrement on allume L2, se programme

```
//C12Autrement.ino
#include "Clair.h"
void setup() { SetupClair();}

void  loop() {
 Si(PousG) {LedGOn;  LedDOff;}
 Autrement {LedGOff; LedDOn; }
}
```

En C	`Si(){}` est exactement le `if(){}` `Autrement{}` est le `else{}` La structure `if() else if() else` est à éviter (voir switch-case).

2 Délai

Il faut pouvoir dire au microcontrôleur d'attendre, pour se mettre au rythme de l'homme ou de la machine qu'il commande. La fonction DelMs(v16); bloque complètement le processeur pendant v16 microsecondes (v16 représente un nombre de 16 bits, 0 à 65535). Quel est le délai max en minutes ?
Note: on parle de fonction quand l'opération demande plusieurs instructions.

Le programme de test suivant simule un ordinateur lent à réagir et permet d'estimer la valeur du temps de réponse que l'on commence à estimer trop long.

```
//C21CopieLente.ino
#include "LcClair.h"
void setup() { SetupLcClair();}

void  loop() {
 Si(PousG) { DelMs(100); LedGOn;}
 Si(!PousG){ DelMs(100); LedGOff;}
}
```

On voit que 100, 200 microsecondes ne sont pas encore gênantes.

Augmentons le délai à 300ms et pressons lentement. Tout est OK.
Pressons rapidement, toutes les secondes environ. La led s'allume irregulierement. Pourquoi ?
Comprendre la programmation temps réel et l'exécution d'un algorithme n'est pas toujour évident !

Expliquons:
Le processeur tourne sans cesse dans la boucle et puisque le poussoir n'est pas pressé, il passe 300ms dans la 2e ligne de code en ignorant ce qui se passe. Si on presse et relâche pendant ce temps, le programme ne l'a pas vu !
Quand le programme quitte le délai de la 2e ligne, il lit le poussoir, c'est instantané. S'il est pressé, ok, on passe dans le délai et on allume après 300ms. En moyenne, en pressant rapidement, le contact est fermé pendant 150ms. On a donc une chance sur 2 quand on presse que le processeur exécute le `if(P1)` quand c'est pressé. Pressez toutes les 2-3 secondes pour ne pas être en phase avec le cycle de 300 ms du programme et mettez des coches pour vérifier cette statistique de 50%.

On change un peu le programme, le délai est après. Qu'est-ce que cela change?

```
void  loop() {
  if(PousG) { LedGOn;  DelMs(100); }
  if(!PousG){ LedGOff; DelMs(100); }
}
```

On change en supprimant le délai de relâchement, ou d'enclenchement. Analyser différents cas donne toujours de l'expérience et permet de choisir les meilleures solutions.

Clignoter

C'est le "Blink" du début, pas si facile en fait. Si on est curieux, on essaie des trucs.

```
//C22Cli.ino
#include "Clair.h"
void setup() { SetupClair();}

void  loop() {
  LedGOn; DelMs(100);
  LedGOff; DelMs(100);
}
```

Truc #1 On met le délai minimum, 1 milliseconde.

Intéressant, on ne voit plus le clignotement, sauf si on bouge l'Edu-C. On a 50% d'intensité lumineuse, mais notre œil dit plus de 50%, il a une réponse logarithmique. Allumez la 2e Led pour comparer.

```
void loop() {
  LedDOn;
  LedGOn; DelMs(100);
  LedGOff; DelMs(100);
}
```

Truc #2 On supprime les instructions de délai

```
void loop() {
  LedGOn;
  LedGOff;
}
```

Tiens, on devrait avoir 50% comme avant, mais c'est très pâle. J'inverse les 2 lignes, Off avant le On, et c'est très lumineux! Bizarre.
La raison est que quand la 2e instruction est exécutée, il faut retourner à la première. Le compilateur a généré une dizaine d'instructions pour traduire le loop(). Donc on aura ~10% allumé ou éteint dans la première instruction. L'effet de la 2e instruction dure pendant le loop.
Truc #3 On utiliser l'instruction **`LedGToggle;`** qui économise une ligne de code, et garantit du 50%.

```
void  loop() {
  LedGToggle; DelMs(100);
}
```

Truc #4 On met le délai à 1 ms et on remplace **`LedGToggle;`** par **`HpToggle;`**

Autre exemple

Une Led doit clignoter 2 fois plus vite que l'autre. On verra plus loin comment leur donner des vitesses différentes.

```
//C23Cli2Leds.ino
#include "Clair.h"
void setup() { SetupClair();}

void loop() {
  LedGToggle; DelMs(200);
  LedDToggle;
  LedGToggle; DelMs(200);
  }
}
```

3 AttendreQue

On a vu que le **`Si()`** teste, fait si, et passe plus loin. Pour attendre que l'on presse ou ait fini de presser sur un poussoir, il faut une nouvelle instruction qui surveille en boucle si l'action voulue arrive; on se bloque dans cette attente. Cela peut être une action extérieure ou un compteur de temps ou d'évènements.

Par exemple on veut allumer/éteindre la Led comme on le fait avec un interrupteur à poussoir domestique qui a un élément basculant interne.

Tant que l'on ne presse pas, il ne se passe rien.
Si on presse on change l'état allumé/éteint.
Tant que l'on presse, il ne se passe rien.
Si on relâche, il ne se passe rien.
On recommence

On peut dire plus simplement: on attend que P1 soit pressé, on change, on attend que P1 soit relâché, et on recommence. Ceci se programme avec l'instruction **`AttendreQue(condition);`**

```
//C31Bascule.ino
#include "Clair.h"
void setup() { SetupClair();}
void  loop() {
  AttendreQue(PousG);
  LedGTog;
  AttendreQue(!PousG); // attend que l'on relâche
}
```

En C	`AttendreQue (vrai){instructions;}` est exactement le `while (!vrai) {instructions;}` On attend tant que la condition contraire est vraie.

4 RepeterSi `RepeterSi (vrai){ instructions;}`

On veut que PousG soit copié sur PousG seulement si le 2e poussoir PousD est pressé (si c'était de la doc Arduino, on mettrait un grand titre "Sécurité nucléaire"). Donc tant que PousD est pressé, on autorise l'exécution du groupe d'instructions que l'on a vu en C11Copie.ino

```
//C41CopieSiOk.ino
#include "Clair.h"
void setup() { SetupClair();}
void  loop() {
  RepeterSi (PousD) {
    if(PousG) { LedGOn; }
    if(!PousG){ LedGOff;}
  }
}
```

Le décalage à chaque ouverture d'accolade est très important pour montrer la structure.

On voit que si on relâche PousD avant PousG, LedG reste allumée. C'est bien l'intention de celui qui a écrit le programme? Pensez à une application industrielle. Si le but de PousD est d'autoriser le fonctionnement on/off, il faut mettre LedGOff quand on relâche P2. Si le but est de ne pouvoir enclencher/déclencher qu'avec l'autorisation, alors c'est le bon programme.

En C	`RepeterSi (vrai){ instructions;}` est exactement le `while (1) {instructions;}`

5 Boucler `Boucler { instructions exécutées en boucle }`

On veut clignoter LedG avant de répéter la copie de PousG sur LedG. On a donc une boucle dans la boucle principale loop().

```
//C51CliEtCopie.ino
#include "Clair.h"
void setup() { SetupClair();}
void  loop() {
  LedDOn; DelMs(200); LedDOff; DelMs(200);
  LedDOn; DelMs(200); LedDOff; DelMs(200);
  Boucler {
     if(PousG) { LedGOn; }
     if(!PousG){ LedGOff;}
  }
}
```

En C	`Boucler {}` est exactement le `while (1) {instructions exécutées en boucle}`

6 Fini `Fini;`

On ne peut pas arrêter le processeur! Il tourne toujours à exécuter l'instruction suivante. Si on veut qu'il ne fasse plus des instructions que l'on a préparé, il faut lui dire de refaire sans cesse l'instruction ou il boucle sur lui-même. C'est l'instruction Fini; que l'on verra plus loin dans un exemple.

En C	`Fini;` est comme `while(1);` ou `while(1){}`

Rappel important

L'instruction **`Si(vrai){xx}`** est dite traversante. On teste, si la condition est vraie, on exécute xx et on passe plus loin.
`RepeterSi(vrai){xx}` est dit bloquant: on teste la condition et on fait et refait xx tant que la condition est vraie.
`AttendreSi(vrai);` est bloquant: on attend que la condition soit vraie (on quitte quant elle est vraie, contrairement au TantQue).

7 Iterer `Iterer (n fois) {instructions;}`

Répéter un certain nombre de fois est fréquent. La fonction **`Iterer(n);`** a besoin d'un paramètre n qui est un nombre ou une variable de 8 bits
Si c'est pour clignoter, il y a alors tout avantage à utiliser le "Toggle" qui inverse l'état. On a 2 fois moins d'instructions qu'il faut répéter 2 fois plus.

```
//C71Cli3fois.ino
#include "Clair.h"
void setup() { SetupClair();}
void  loop() {
  Iterer (3*2) {
    LedGToggle; DelMs(200);
  }
  DelMs (1000);
}
```

Au lieu du **`DelMs(1000);`** ou pourrait mettre le **`Fin;`** que l'on vient de voir, pour ne faire qu'une fois. Il faut alors presser sur le bouton reset pour redémarrer le programme, et l'environnement Arduino exécute des milliers d'instructions avant (avec beaucoup de tests et de boucles d'attente).
Vous avez compris pourquoi c'est important de noter 3*2 et pas 6 pour dire combien de "Toggle"?

SOS

En passant, on pourrait envoyer un SOS `... --- ...`

```
//C72SOS.ino On envoie 2 fois et on stoppe
#include "Clair.h"
void setup() { SetupClair();}

void  loop() {
  Repete(2) {
    Repete(3*2) { LedGToggle; DelMs(200); }
    DelMs(500);
    Repete(3*2) { LedGToggle; DelMs(600); }
    DelMs(500);
    Repete(3*2) { LedGToggle; DelMs(200); }
  }
  Fin;
```

```
}
```

Encore une astuce. Pour redémarrer le programme, au lieu de `Fin;` on pourrait mettre `AttendreQue(PousG){} ou AttendreQue(PousG);`
Une accolade vide veut dire que l'on ne fait rien. Un ; seul dit la même chose.
On recommencera après loop() sans passer par le setup, ce sera plus rapide que le reset.

En C	`Iterer(n){}` est exactement le `for (byte i=0; i<n; i++) {faire et refaire}` La structure `for(){}` est très riche, mais on l'utilise surtout pour répéter

8 Echapper `Echapper;` (pour avançés)

Imaginons que l'on est dans un programme qui clignote sans cesse. Pour en sortir si on presse sur PousG, il y a l'instruction **`Echapper;`** , qui permet de s'échapper de ce qui se passe dans l'accolade associée à un RepeterSi, Iterer ou Boucler. Cela sera très utile pour des jeux: il se passe des choses sur les Leds, sur l'écran et il faut réagir. Avec Echapper; on sort "en catastrophe" de cet ensemble d'activité qui modifiait des variables, des compteurs, et on décide dans le module suivant si c'est bien joué.
Comme exemple simple, on clignote en permanence LedG. PousD interrompt le clignotement; il éteint LedG, allume LedD et stoppe.

```
 //C81EchappeCli.ino
#include "Clair.h"
void setup() { SetupClair();}

void  loop() {
  Boucler {
    LedGToggle; DelMs(200);
    if (PousG) {LedGOff; Echapper;}
  }
  LedDOn;
  Fin;
}
```

En C	`Echapper;` est exactement le `break;` C'est une instruction qui a plusieurs rôles, délicate à bien utiliser

9 Nombres, variables et calculs

Il y a plusieurs notions délicates associées aux nombres et variables. Restons simple.
Les nombres sont des entiers positifs, mais on ne peut pas écrire n'importe quelle valeur. Il y a des limitations à 8 bits (0 à max 255), à 16 bits (max 65535) selon la fonction qui va utiliser le nombre.
Une variable est une case, un tiroir, une enveloppe vide avec un nom. Elle contient un octet, un nombre entier 8 bits appelé **`byte`**, avec une valeur entre 0 et 255. On déclare en écrivant
`byte toto;` ce qui réserve une position mémoire que le programme pourra lire et écrire.
Pour avoir des nombres plus grands, on déclare int (16 bits) et long (32 bits) et le compilateur réserve plus de mémoire.
On peut calculer avec des nombres et variables, mais le résultat est ramené dans la taille possible, le compilateur C ne sait pas ce que vous cherchez à faire et ne vérifie pas.
Les opérations sont + - * / % et il faut mettre des parenthèses pour savoir dans quel ordre les calculs sont effectués. La division en nombre entier donne un quotient et un reste. / donne le quotient % donne le reste.
On peut comparer des nombres et variables et décider comme on a fait avec les poussoirs.
Les opérations sont == >= <= != (puisque ! est l'inversion, le dernier veut dire différent).
Pour vérifier, on peut faire un petit programme qui divise, recalcule et compare.

```
//C91Division.ino
#include "Clair.h"
void setup() { SetupClair();}
  byte nombre,diviseur, quotient,reste;
void  loop() {
   quotient = nombre/diviseur;
   reste = nombre%diviseur;
   Si((quotient*diviseur)+reste==nombre){ LedGOn;}
   Autrement {LedDOn;}
   Fin;
}
```

Evidemment sous cette forme, le résultat sera toujours correct. Mettons des nombres (calculés par le programmeur) et demandons au programme de vérifier.

```
//C92VerifDivision.ino
#include "Clair.h"
void setup() { SetupClair();}
  byte quotient,reste;
  #define Nombre=52
  #define Diviseur=5
void  loop() {
   quotient = Nombre/Diviseur;
   reste = Nombre%Diviseur;
   Si((quotient*5)+reste==52){ LedGOn;}
   Autrement {LedDOn;}
   Fin;
}
```

Note; ces notions seront reprises avec affichage sur oled.

10 Faire plusieurs choses en même temps

PousG déclenche une alarme, la LedG allumée pendant 10 secondes. On veut pouvoir quittancer l'alarme, éteindre avec PousD, avant les 10 secondes. DelMs(10000); est bloquant, le processeur ne fait rien d'autre que d'attendre. Il faut lui demander de remplacer sa grande attente par plusieurs petites, comme cela on peut demander au processeur de faire autre chose, ici surveiller PD.
Pour ne pas rater l'action d'un utilisateur humain sur PD, il faut le lire toutes les 0.1 secondes. On va donc remplacer le délai de 10 secondes par 100 délais de 0,1s, soit 100ms.

```
//C101Alarme.ino 10s LedGOn coupé par Pous1
#include "Clair.h"
void setup() { SetupClair();}
  byte cnt;
void  loop() {
  LedGOn; cnt=0; // pour 10 secondes
  TantQue (cnt<100) {
    DelMs(100); cnt++;     // On compte les cycles
    if (PousG) {cnt=100;} // force la fin
  }
  LedGOff;
  DelMs(2000); // 2s d'attente avant de recommencer
}
```

On voit que, en allumant la LedG, on a initialisé un compteur de cycles qui va compter toutes les 0.1s=100ms. On reste dans la boucle tant que le compteur est inférieur à 100. En même temps (l'instruction suivante), on teste PousG et s'il est pressé, on force la fin.

Ce qui est intéressant avec ce programme, c'est qu'il fait deux choses "en même temps".

Vous voyez comment le modifier pour faire un premier jeu de réflexe? Quand la Led s'allume, il faut l'éteindre le plus vite possible. On peut allumer la LedD si le temps est bon.

Deux Leds qui clignotent

On veut que les deux Leds clignotent à des vitesses différentes (on devrait dire fréquence, mais c'est la période que l'on va changer). Dans une boucle de 10ms, on met 2 compteurs et on surveille leur valeur. Si cette valeur dépasse une consigne (valeur maximale), on redémarre le compteur et on clignote (change l'état de la led correspondant au compteur).

```
//C44CliDif.ino Clignote à des périodes différentes
#include "Clair.h"
void setup() { SetupClair();}
int cntG,cntD;  // variables délais en 16 bits
#define PerG 189
#define PerD 190
void  loop() {
   DelMs (1);
   if (cntG++ > PerG) { cntG=0; LedGToggle; }
   if (cntD++ > PerD) { cntD=0; LedDToggle; }
}
```

On va naturellement essayer d'autres valeurs. Le PGCD, vous l'avez vu à l'école? C'est lié au temps pour que les Leds soient "en phase".
Remplacez LedDToggle par LedGToggle (la même led) et mettez 18 19 pour les temps. Vous comprenez ce qui se passe?

11 Hasard et jeux simples

La fonction Alea(min,max) génère un nombre entre min et max. C
Voir www.didel.com/educ/EduC-Mod3 pour des exemples, faciles à compléter.

12 Pour continuer

A ce stade, on peut programmer des jeux de réflexe, faire des mélodies et se limiter à la fonctionnalité du LCbot (voir LcClair.pdf).
On peut continuer et documenter les potentiomètres et l'écran Oled, avec les mêmes programmes Fun ou EduC convertis pour utiliser les instructions de Clair.h.
On peut continuer avec Fun, éventuellement adapté pour suivre une filière ***Clair*** avec graphique.
On peut recommencer avec les Modules EduC et utiliser la librairie EduC.h qui force à programmer en C.

jdn 180227/181205