Langage C

arduino-premiers-pas-en-informatique-embarquee

Arduino - Référence EN

Le grand Livre Arduino

Apprenez-a-concevoir-avant-de-programmer

Histoire et évolution des langages de programmation

Apprenez à programmer en C 2e Édition

Le Langage C

La POO et le langage arduino

Pourquoi apprendre à coder

Le codage, qu'est-ce que c'est ? Il s'agit simplement du langage de programmation informatique. Maîtriser cette nouvelle syntaxe permet de créer des actions et de commander aux machines. Un graphiste peut par exemple concevoir un site Internet, s'il sait coder il est également capable de lui donner vie sur la toile.

Maxime de Nicolas Boileau Philosophe du 17 eime siecle :

- *"Avant donc que d'écrire, apprenez à penser."*
- *"Ce que l'on conçoit bien s'énonce clairement."*
- *"Et les mots pour le dire arrivent aisément".*
- *"Hâtez-vous lentement, et sans perdre courage,"*
- *"Vingt fois sur le métier remettez votre ouvrage."*
- *"Polissez-le sans cesse, et le repolissez."*
- *"Ajouter quelquefois et souvent effacez."*

Savoir programmer permet ainsi de :

1. Grossir son CV : savoir combiner cette double casquette constitue une véritable plus-value sur le marché de l'emploi. Le code enrichit vos compétences, vous rend plus précieux, plus utile, voire indispensable. Bref, vous sortez du lot.

2. Comprendre son environnement : jeux vidéo, applications, pages web… être initié au code c'est savoir décrypter les rouages de ces objets qui nous gouvernent.

3. Partager : contrairement aux idées reçues, le coding est régi par un esprit altruiste, c'est le fameux « open source ». Vous pouvez copier des codes existants mais aussi offrir vos créations. Bienvenue au programme des bisounours.

4. Aller au bout de vos idées : informé du champ des possibles, vous connaîtrez les contraintes pour mettre en œuvre vos concepts et ainsi vous assurer de leur aboutissement.

5. Simplifier les process : grâce à cette nouvelle compétence, un client peut vous confier un projet sans avoir recours à d'autres prestataires. Vous devenez ainsi l'interlocuteur dédié, c'est un gain de temps et de réactivité, et donc d'argent.

6. Inventer des solutions opérationnelles : savoir coder, permet de concevoir des créations réalistes, vouées à exister.

7. Mieux communiquer : connaître le code c'est aussi maîtriser les bons éléments de langage pour expliquer vos idées à un intégrateur ou un développeur. Et d'optimiser ainsi la gestion de votre projet.

8. Booster sa créativité : l'univers du code n'est pas uniquement technique, il offre de nombreuses opportunités créatives. Il s'agit de concevoir des programmes, mais aussi d'imaginer comment leur donner vie.

9. Se la raconter : le codage a le vent en poupe, le succès de l'Ecole 42 en témoigne (Lire aussi L'Ecole 42, élue meilleure Code Factory). Vous commandez aux machines, vous êtes dans la matrice, c'est jubilatoire.

10. S'amuser : l'apprentissage du code consiste à comprendre une nouvelle langue, son alphabet, sa grammaire, ses modes d'expression… afin de créer des actions. Même les plus jeunes s'y mettent et peuvent créer leurs propres jeux grâce notamment aux Magic Makers ou aux coding-goûters.

Apprendre la programmation

Les Données

Les données manipulées en langage C sont typées, c'est-à-dire que pour chaque donnée que l'on utilise (dans les variables par exemple) il faut préciser le type de donnée, ce qui permet de connaître l'occupation mémoire (le nombre d'octets) de la donnée ainsi que sa représentation :

- des nombres : entiers (int) ou réels, c'est-à-dire à virgules (float)
- des pointeurs (pointer) : permettent de stocker l'adresse d'une autre donnée, ils « pointent » vers une autre donnée

En C il existe plusieurs types entiers, dépendant du nombre d'octets sur lesquels ils sont codés ainsi que de leur format, c'est-à-dire s'ils sont signés (possédant le signe - ou +) ou non. Par défaut les données sont signées.

Voici un tableau donnant les types de données en langage C :

Type de donnée	Signification	Taille (en octets)	Plage de valeurs acceptée
char	Caractère	1	-128 à 127
unsigned char	Caractère non signé	1	0 à 255
short int	Entier court	2	-32 768 à 32 767
unsigned short int	Entier court non signé	2	0 à 65 535
int	Entier	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	-32 768 à 32 767 -2 147 483 648 à 2 147 483 647
unsigned int	Entier non signé	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	0 à 65 535 0 à 4 294 967 295

Type de donnée	Signification	Taille (en octets)	Plage de valeurs acceptée
long int	Entier long	4	-2 147 483 648 à 2 147 483 647
unsigned long int	Entier long non signé	4	0 à 4 294 967 295
float	Flottant (réel)	4	3.4*10-38 à 3.4*1038
double	Flottant double	8	1.7*10-308 à 1.7*10308
long double	Flottant double long	10	3.4*10-4932 à 3.4*104932

Les Variables

langage-c-les-variables

Types et tailles des données exemple : int8_t

Il est dans certains cas nécessaire d'utiliser des types de données qui donnent les mêmes intervalles de valeurs sur tous les ordinateurs. La norme ISO C99 impose de définir des types portables afin de régler ces problèmes sur toutes les architectures existantes. Ces types sont définis dans le fichier d'en-tête stdint.h. Il s'agit des **types int8_t, int16_t, int32_t et int64_t**, et de leurs versions **non signées uint8_t, uint16_t, uint32_t et uint64_t.** La taille de ces types en bits est indiquée dans leur nom et leur utilisation ne devrait pas poser de problème.

Tableaux de chaines de caractères

Une chaîne de caractères est un tableau à 1 dimension de caractères.

On peut également définir des tableaux à plusieurs dimensions qui peuvent contenir des mots:(le tableau commence à 0)

char JOUR[7][9] = {“lundi” , ”mardi” , ”mercredi” , ”jeudi” ,”vendredi”,”samedi”,”dimanche”}; [7] ⇒ il y a 7 jours , [9] ⇒ 9 caractères au maximum par mots

JOUR[0] = “lundi” etc ..

et on peut accéder à ces mots en utilisant la syntaxe suivante: int I=2; Serial.println(“Aujourd’hui nous sommes”,JOUR[I]);

qui affichera “Aujourd’hui nous sommes mercredi”

la fonction String

l'objet String. Qu'est-ce qu'un objet?

Un objet est une construction qui contient à la fois des données et des fonctions. Un objet String peut être créé comme une variable et assigné une valeur ou une chaîne. L'objet String contient des fonctions (appelées «méthodes» en programmation orientée objet (POO)) qui opèrent sur les données de chaîne contenues dans l'objet String.

L'esquisse et l'explication suivantes expliquent clairement ce qu'est un objet et comment l'objet String est utilisé. Exemple :

exemple1.ino

```
void setup() {
   String my_str = "This is my string.";
   Serial.begin(9600);

   // (1) print the string
   Serial.println(my_str);

   // (2) change the string to upper-case
   my_str.toUpperCase();
   Serial.println(my_str);

   // (3) overwrite the string
   my_str = "My new string.";
   Serial.println(my_str);

   // (4) replace a word in the string
   my_str.replace("string", "Arduino sketch");
   Serial.println(my_str);

   // (5) get the length of the string
   Serial.print("String length is: ");
   Serial.println(my_str.length());
}

void loop() {

}
```

Résultat

```
This is my string.
THIS IS MY STRING.
My new string.
My new Arduino sketch.
String length is: 22
```

Un objet chaîne est créé et une valeur (ou chaîne) est attribuée en haut de l'esquisse.

```
String my_str = "This is my string." ;
```

Cela crée un objet String avec le nom my_str et lui donne la valeur “Ceci est ma chaîne.”.

Cela peut être comparé à la création d'une variable et à lui attribuer une valeur telle qu'un entier -

```
int my_var = 102;
```

L'esquisse fonctionne de la manière suivante. Impression de la chaîne

La chaîne peut être imprimée dans la fenêtre Serial Monitor comme une chaîne de tableau de caractères. Convertir la chaîne en majuscules

L'objet chaîne my_str qui a été créé a un certain nombre de fonctions ou de méthodes qui peuvent être utilisées. Ces méthodes sont appelées en utilisant le nom des objets suivi de l'opérateur point (.) Puis du nom de la fonction à utiliser.

```
my_str.toUpperCase();
```

La fonction toUpperCase () opère sur la chaîne contenue dans l' objet my_str qui est de type String et convertit les données de chaîne (ou texte) que l'objet contient en caractères majuscules. Une liste des fonctions que la classe String contient peut être trouvée dans la référence Arduino String. Techniquement, String est appelé une classe et est utilisé pour créer des objets String. Écraser une chaîne

L'opérateur d'affectation est utilisé pour affecter une nouvelle chaîne à l' objet my_str qui remplace l'ancienne chaîne

```
my_str = "My new string." ;
```

L'opérateur d'affectation ne peut pas être utilisé sur les chaînes de tableau de caractères, mais fonctionne uniquement sur les objets String. Remplacement d'un mot dans la chaîne

La fonction replace () est utilisée pour remplacer la première chaîne qui lui est passée par la deuxième chaîne qui lui est passée. replace () est une autre fonction intégrée à la classe String et donc disponible pour une utilisation sur l'objet String my_str. Obtenir la longueur de la chaîne

Obtenir la longueur de la chaîne se fait facilement en utilisant length (). Dans l'exemple d'esquisse, le résultat renvoyé par length () est passé directement à Serial.println () sans utiliser de variable intermédiaire. Quand utiliser un objet chaîne

Un objet String est beaucoup plus facile à utiliser qu'un tableau de caractères chaîne. L'objet a des fonctions intégrées qui peuvent effectuer un certain nombre d'opérations sur des chaînes.

Le principal inconvénient de l'utilisation de l'objet String est qu'il utilise beaucoup de mémoire et peut rapidement utiliser la mémoire RAM Arduinos, ce qui peut entraîner le blocage, le blocage ou un comportement inattendu d'Arduino. Si un croquis sur un Arduino est petit et limite l'utilisation d'objets, il ne devrait y avoir aucun problème.

Les chaînes de tableaux de caractères sont plus difficiles à utiliser et vous devrez peut-être écrire vos propres fonctions pour opérer sur ces types de chaînes. L'avantage est que vous avez le contrôle sur la taille des tableaux de chaînes que vous créez, vous pouvez donc garder les tableaux petits pour économiser de la mémoire.

Vous devez vous assurer que vous n'écrivez pas au-delà de la fin des limites du tableau avec des tableaux de chaînes. L'objet String n'a pas ce problème et s'occupera des limites de chaîne pour vous, à condition qu'il y ait suffisamment de mémoire pour qu'il puisse fonctionner. L'objet String peut essayer d'écrire dans la mémoire qui n'existe pas lorsqu'il manque de mémoire, mais n'écrira jamais sur la fin de la chaîne sur laquelle il fonctionne. Où les chaînes sont utilisées

Dans ce chapitre, nous avons étudié les chaînes, leur comportement en mémoire et leurs opérations.

Les utilisations pratiques des chaînes seront couvertes dans la partie suivante de ce cours lorsque nous étudierons comment obtenir une entrée utilisateur à partir de la fenêtre Serial Monitor et enregistrer l'entrée dans une chaîne.

Les structures en langage C

Différence entre une structure et un tableau

Un tableau permet de regrouper des éléments de même type, c'est-à-dire codés sur le même nombre de bits et de la même façon. Toutefois, il est généralement utile de pouvoir rassembler des éléments de type différent tels que des entiers et des chaînes de caractères.

Les structures permettent de remédier à cette lacune des tableaux, en regroupant des objets (des variables) au sein d'une entité repérée par un seul nom de variable.

Les objets contenus dans la structure sont appelés champs de la structure. Déclaration d'une structure

Lors de la déclaration de la structure, on indique les champs de la structure, c'est-à-dire le type et le nom des variables qui la composent :

```
struct Nom_Structure {
 type_champ1 Nom_Champ1;

 type_champ2 Nom_Champ2;

 type_champ3 Nom_Champ3;

 type_champ4 Nom_Champ4;

 type_champ5 Nom_Champ5;

 ...
};
```

La dernière accolade doit être suivie d'un point-virgule !

Le nom des champs répond aux critères des noms de variable

Deux champs ne peuvent avoir le même nom

Les données peuvent être de n'importe quel type hormis le type de la structure dans laquelle elles se trouvent

Ainsi, la structure suivante est correcte :

```
struct MaStructure {
 int Age;
```

```
 char Sexe;

 char Nom[12];

 float MoyenneScolaire;

 struct AutreStructure StructBis;
/* en considérant que la structure AutreStructure est définie */
};
```

Par contre la structure suivante est incorrecte :

```
struct MaStructure {
 int Age;

 char Age;

 struct MaStructure StructBis;

};
```

Il y a deux raisons à cela :

- Le nom de variable Age n'est pas unique
- Le type de donnée struct MaStructure n'est pas autorisé

La déclaration d'une structure ne fait que donner l'allure de la structure, c'est-à-dire en quelque sorte une définition d'un type de variable complexe. La déclaration ne réserve donc pas d'espace mémoire pour une variable structurée (variable de type structure), il faut donc définir une (ou plusieurs) variable(s) structurée(s) après avoir déclaré la structure... Définition d'une variable structurée

La définition d'une variable structurée est une opération qui consiste à créer une variable ayant comme type celui d'une structure que l'on a précédemment déclarée, c'est-à-dire la nommer et lui réserver un emplacement en mémoire.

La définition d'une variable structurée se fait comme suit :

```
  struct Nom_Structure Nom_Variable_Structuree;
```

Nom_Structure représente le nom d'une structure que l'on aura préalablement déclarée.

Nom_Variable_Structuree est le nom que l'on donne à la variable structurée.

Il va de soi que, comme dans le cas des variables on peut définir plusieurs variables structurées en les séparant avec des virgules :

```
  struct Nom_Structure Nom1, Nom2, Nom3, ...;
```

Soit la structure Personne :

```
struct Personne{
```

```
  int Age;

  char Sexe;

};
```

On peut définir plusieurs variables structurées :

```
  struct Personne Pierre, Paul, Jacques;
```

Accès aux champs d'une variable structurée

Chaque variable de type structure possède des champs repérés avec des noms uniques. Toutefois le nom des champs ne suffit pas pour y accéder étant donné

qu'ils n'ont de contexte qu'au sein de la variable structurée...

Pour accéder aux champs d'une structure on utilise l'opérateur de champ (un simple point) placé entre le nom de la variable structurée que l'on a défini et le nom du champ :

```
  Nom_Variable.Nom_Champ;
```

Ainsi, pour affecter des valeurs à la variable Pierre (variable de type struct Personne

définie précédemment), on pourra écrire :

Pierre.Age = 18;

Pierre.Sexe = 'M';

Tableaux de structures

Etant donné qu'une structure est composée d'éléments de taille fixe, il est possible de créer un tableau ne contenant que des éléments du type d'une structure donnée. Il suffit de créer un tableau dont le type est celui de la structure et de le repérer par un nom de variable :

```
  struct Nom_Structure Nom_Tableau[Nb_Elements];
```

Chaque élément du tableau représente alors une structure du type que l'on a défini...

Le tableau suivant (nommé Repertoire) pourra par exemple contenir 8 variables structurées de type struct Personne :

```
  struct Personne Repertoire[8];
```

De la même façon, il est possible de manipuler des structures dans les fonctions.

L'instruction if

L'instruction if est la structure de test la plus basique, on la retrouve dans tous les langages (avec une

syntaxe différente...). Elle permet d'exécuter une série d'instructions si jamais une condition est réalisée.

La syntaxe de cette expression est la suivante :

```
if (condition réalisée) {
liste d'instructions;

}
```

Remarques :

- la condition doit être entre des parenthèses
- il est possible de définir plusieurs conditions à remplir avec les opérateurs ET et OU (&& et ||)

Par exemple l'instruction suivante teste si les deux conditions sont vraies :

```
if ((condition1)&&(condition2))
```

L'instruction suivante exécutera les instructions si l'une ou l'autre des deux conditions est vraie :

```
if ((condition1)||(condition2))
```

- s'il n'y a qu'une instruction, les accolades ne sont pas nécessaires...
- les instructions situées dans le bloc qui suit else sont les instructions qui seront exécutées si la ou les conditions ne sont pas remplies

L'instruction if ... else

L'instruction if dans sa forme basique ne permet de tester qu'une condition, or la plupart du temps on aimerait pouvoir choisir les instructions à exécuter en cas de non réalisation de la condition...

L'expression if ... else permet d'exécuter une autre série d'instructions en cas de non-réalisation de la condition.

La syntaxe de cette expression est la suivante :

```
if (condition réalisée) {
liste d'instructions
}

else {
autre série d'instructions
}
```

Une façon plus courte de faire un test

Il est possible de faire un test avec une structure beaucoup moins lourde grâce à

la structure suivante :

(condition) ? instruction si vrai : instruction si faux</i>

Remarques :

- la condition doit être entre des parenthèses
- Lorsque la condition est vraie, l'instruction de gauche est exécutée
- Lorsque la condition est fausse, l'instruction de droite est exécutée
- En plus d'être exécutée, la structure ?: renvoie la valeur résultant de l'instruction exécutée. Ainsi, cette forme ?: est souvent utilisée comme suit :

position = ((enAvant == 1) ? compteur+1 : compteur-1);

L'instruction switch

L'instruction switch permet de faire plusieurs tests de valeurs sur le contenu d'une même variable. Ce branchement conditionnel simplifie beaucoup le test de plusieurs valeurs d'une variable, car cette opération aurait été compliquée (mais possible) avec des if imbriqués. Sa syntaxe est la suivante :

```
switch (Variable) {

case Valeur1 :
Liste d'instructions;
break;

case Valeur2 :
Liste d'instructions;
break;

case Valeurs... :
Liste d'instructions;
break;

default:
Liste d'instructions;

}
```

Les parenthèses qui suivent le mot clé switch indiquent une expression dont la valeur est testée successivement par chacun des case. Lorsque l'expression testée est égale à une des valeurs suivant un case, la liste d'instructions qui suit celui-ci est exécutée. Le mot clé break indique la sortie de la structure conditionnelle. Le mot clé default précède la liste d'instructions qui sera exécutée si l'expression n'est jamais égale à une des valeurs.

- *N'oubliez pas d'insérer des instructions break entre chaque test, ce genre d'oubli est difficile à détecter car aucune erreur n'est signalée... En effet, lorsque l'on omet le break, l'exécution continue dans les blocs suivants ! **

Cet état de fait peut d'ailleurs être utilisé judicieusement afin de faire exécuter les mêmes instructions pour différentes valeurs consécutives, on peut ainsi mettre plusieurs cases avant le bloc :

```
switch(variable)
{
case 1:
case 2:
{ instructions exécutées pour variable = 1 ou pour variable = 2 }
break;
case 3:
{ instructions exécutées pour variable = 3 uniquement }
break;
default:
{ instructions exécutées pour toute autre valeur de variable }
}
```

Les boucles

Les boucles, dans un programme, permettent de répéter certaines instructions plusieurs fois sans avoir à recoder plusieurs fois ces instructions. En C il existe trois types de boucles, nous parlerons de chacune d'elle. Les voici :

- **for**
- **while**
- **do / while**

La boucle for

La boucle for teste une condition avant d'exécuter les instructions qui lui sont associées. Voilà sa synthaxe :

```
for (expression1 ; expression2 ; expression3)
  {
        instructions à réaliser
  }
```

expression1 sera une initialisation d'une variable dit de contrôle qui servira lors du test de condition de réalisation des instructions de la boucle.

expression2 ce sera justement la condition pour que la boucle s'exécute.

expression3 permettra de modifier la valeur de la variable de contrôle à chaque passage de la boucle.

N'oubliez pas de séparer ces 3 expressions par un “ ; ”

Voyons maintenant son fonctionnement à travers un programme qui se chargera d'afficher 3 fois le terme “Hello world!”

```
int main()
{
        int i = 0; /* voilà notre variable de contrôle */
        for ( i = 0 ; i < 3 ; i++)
```

```
            {
                printf("Hello World !\n");
            }
        return 0;
  }
```

En testant ce code vous devriez avoir quelque chose comme :

Code console:

Hello World!

Hello World!

Hello World!

Détaillons maintenant le code que nous venons de créer :

```
 for (i = 0 ; i < 3 ; i++)
    i = 0 ; //On commence par initialiser notre variable de contrôle à 0 ( ne pas oublier de la déclarer avant dans le programme ? )
    i < 3 ; //C'est la condition pour que la boucle s'exécute, tant que i sera inférieur 3 les instructions à l'intérieur de la boucle continuerons de s'exécuter
    i++ ; //(équivalent à i + 1 [dorénavant préférer le ++ plutôt que le + 1 sur une variable ou le -- plutôt que le - 1]) Cette expression signifie qu'on ajoutera 1 à la variable i à chaque passage de boucle.
```

Voilà comment on pourrait traduire cette boucle en français :

Commence avec i = 0 ; Faire tant que i < 3 ; Rajoute 1 à chaque passage de la boucle ;

Afficher le message “Hello World!” Reviens au début Rajoute 1 i = 1 donc < 3 On peux recommencer Afficher le message “Hello World!” Reviens au début Rajoute 1 i = 2 donc < 3 On peux recommencer Afficher le message “Hello World! Rajoute 1 i = 3 donc i < 3 est validé Quitte la boucle Fin du programme.

Voilà sommairement ce que fait notre programme. ?

Il n'y a pas de ”;“ a la fin du for c'est une erreur assez répandu chez ceux qui débute donc faites y attention ?

La boucle while

Continuons maintenant avec la boucle while

La boucle while, tout comme la boucle for ne permet l'exécution d'instructions que si une condition vérifiée, avant l'exécution des instructions, est vrai (TRUE). Voici sa synthaxe :

```
while ( condition )
```

```
{
    instructions(s)
}
```

voyons son fonctionnement à travers le même programme que tout à l'heure à savoir : afficher trois fois le message “Hello World !”

```
int main()
  {
       int i = 0;
       while ( i < 3 )
       {
               printf("Hello World !\n");
               i++;
        }
        return 0;
  }
```

Détaillions :

On commence comme d'habitude par initialiser notre variable de contrôle (i). Ensuite on rentre dans la boucle car la condition est vérifiée (i vaut 0 donc est bien < à 3) Puis on affiche le fameux hello world!. On ajoute 1 à la variable i. Le programme retourne au début, recommence à exécuté les instructions car àa ce moment i vaut 1 … et ainsi de suite jusqu'à ce que i soit égal à 3 lors du test de condition, ce qui fera quitter la boucle au programme.

Il n'y a pas non plus de ”;“ à la fin du while

Vous commencez à comprendre le principe des boucles ? C'est bien ?

La boucle do / while

Il ne nous en reste plus qu'une à voir : la boucle do / while

La boucle do / while diffère des autres dans le sens où les instructions qui lui sont rattachées s'exécutent au moins une fois. Cela étant du à sa synthaxe :

```
do
{
       instruction(s) à réaliser
}while (condition);
```

Comme vous pouvez le voir la condition pour que la boucle se réalise se situe à la fin de la boucle. Or comme vous le savez déjà peu être, votre ordinateur lit le programme de bas en haut ce qui fait que les instructions d'une boucle do / while s'exécute au moins une fois

Ici ne pas oublier le ” ; “ à la fin du while

petite mise en garde sur les boucles infinie

Utiliser des boucles dans un programme c'est bien, mais si on ne fait pas attention à ce que l'on écrit on peut vite arriver à ce que l'on appel une boucle infinie, c'est à dire une boucle dont la condition est toujours vrai, donc qui ne s'arrêtera jamais et bloquera votre programme. Heureusement les systèmes d'exploitation actuel savent faire face à ce genre de problème et vous n'aurez pas de mal à arrêter votre programme.

Voici des exemples de boucle infinie (à éviter de reproduire donc ?)

```
while (1)
{
     printf("Boucle infinie !");
}

/* 1 est toujours vrai */

for ( i = 0 ; i < -1 ; i++)
{
     printf("Boucle infinie !");
}

/* i vaut 0 au départ donc sera TOUJOURS supérieur a -1 dans cette boucle !
*/

do
{
     printf("Boucle infinie !");
}while (4 < 5);

/* 4 est toujours supérieur à 5 */
```

Soyez attentif et pensez votre code avant de le compiler ?

Conclusion

Voilà vous savez maintenant comment faire une boucle dans un programme C. Pourquoi ne pas vous entrainer en créant un petit programme qui fera autant de tour que l'utilisateur voudra ? et qui affichera le nombre de tours effectués ?

Essayer d'avoir un rendu comme celui ci :

Code console: Combien de tour SVP : 3

Nombre de tours dans la boucle : 1

Nombre de tours dans la boucle : 2

Nombre de tours dans la boucle : 3

Sortie de la boucle...

Ou vous pouvez tout aussi bien tenter de créer une calculette multi-fonctions ou encore des petits programmes de révisions des dates historiques par exemple.

Les fonctions ou sous programmes

les fonctions FR

Les fonctions FR

Dans un programme, les lignes sont souvent très nombreuses. Il devient alors impératif de séparer le programme en petits bouts afin d'améliorer la lisibilité de celui-ci, en plus d'améliorer le fonctionnement et de faciliter le débogage. Nous allons voir ensemble ce qu'est une fonction, puis nous apprendrons à les créer et les appeler.

Qu'est-ce qu'une fonction ?

Une fonction est un "conteneur" mais différent des variables. En effet, une variable ne peut contenir qu'un nombre, tandis qu'une fonction peut contenir un programme entier ! Par exemple ce code est une fonction :

Standardiser les fragments de code en fonctions présente plusieurs avantages :

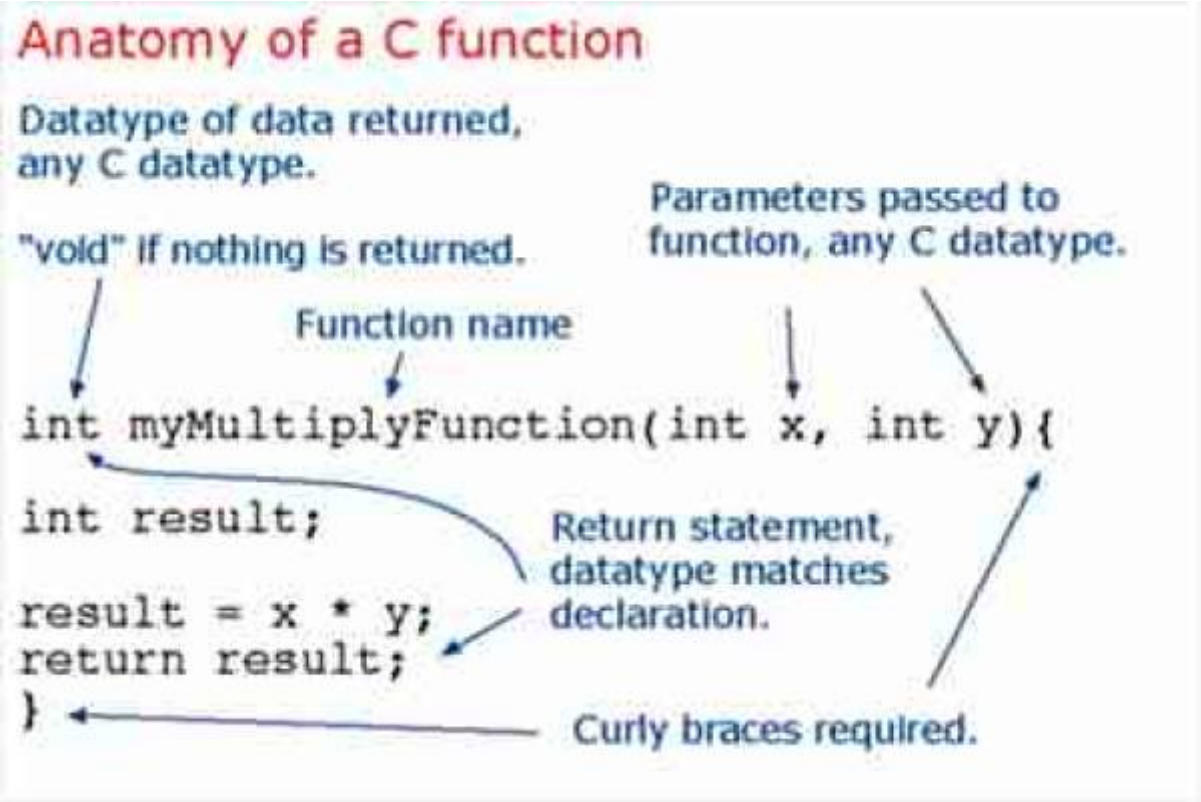
- Les fonctions aident le programmeur à rester organisé. Cela aide souvent à conceptualiser le programme.
- Les fonctions codifient une action en un seul endroit afin que la fonction n'ait à être pensée et déboguée qu'une seule fois.
- Cela réduit également les risques d'erreurs de modification, si le code doit être modifié.

Les fonctions rendent l'ensemble de l'esquisse plus petite et plus compacte car des sections de code sont réutilisées plusieurs fois. Ils facilitent la réutilisation du code dans d'autres programmes en le rendant plus modulaire et, comme effet secondaire agréable, l'utilisation de fonctions rend souvent le code plus lisible.

Il y a deux fonctions requises dans une esquisse Arduino, setup() et loop().

D'autres fonctions doivent être créées en dehors des parenthèses de ces deux fonctions. A titre d'exemple, nous allons créer une fonction simple pour multiplier deux nombres.

Exemples :



Pour “appeler” notre simple fonction de multiplication, nous lui passons les paramètres du type de données qu'elle attend;

Notre fonction doit être déclarée en dehors de toute autre fonction, donc “myMultiplyFunction()” peut aller au-dessus ou au-dessous de la fonction “loop()”.

L'ensemble de l'esquisse ressemblerait alors à ceci :

0001.ino

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

void loop() {
  int i = 2;
  int j = 3;
  int k;

  k = myMultiplyFunction(i, j); // k now contains 6
  Serial.println(k);
  delay(500);
}

int myMultiplyFunction(int x, int y){
  int result;
  result = x * y;
  return result;
```

```
}
```

L'IDE Arduino permet d'appeler une fonction avant sa définition.

Dans les fichiers .cpp, vous devez définir la fonction ou du moins déclarer le prototype de fonction avant de pouvoir l'utiliser. Dans un fichier .ino, l'IDE Arduino crée un tel prototype dans les coulisses.

From:
http://chanterie37.fr/fablab37110/ - **Castel'Lab le Fablab MJC de Château-Renault**

Permanent link:
http://chanterie37.fr/fablab37110/doku.php?id=start:arduino:langage_c&rev=1664436035

Last update: **2023/01/27 16:08**