Mes débuts en programmation Zig

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 3 novembre 2023

https://www.bortzmeyer.org/mes-debuts-en-zig.html

Je suis en train d'apprendre le langage de programmation Zig donc je vous fais profiter ici de ce que j'ai appris. Je ne vais pas détailler les points secondaires, comme la syntaxe, mais plutôt parler des concepts originaux de Zig, comme sa gestion d'erreurs, ses variables optionnelles ou son modèle de gestion mémoire.

Attention, si vous êtes sénateur, ne lisez pas ce texte <https://www.francetvinfo.fr/societe/education/ecriture-inclusive/ecriture-inclusive-cinq-questions-sur-la-proposition-de-loi-6155388.html>, il utilise plusieurs techniques d'écriture inclusive. Le risque de crise cardiaque est particulièrement élevé après un déjeuner bien arrosé au restaurant du Sénat.

D'abord, le cahier des charges. Zig est un langage de bas niveau, au sens où il est conçu pour des programmes où le·a programmeur·se contrôle plein de détails, comme l'allocation de mémoire. Il est prévu pour programmer sur le métal nu (par exemple programmation dite « embarquée » ou bien pour faire un noyau de système d'exploitation). Mais on peut aussi l'utiliser sur un environement plus perfectionné, par exemple sur Unix pour écrire des serveurs Internet (le domaine de la programmation que je connais le mieux).

Zig est donc un concurrent de C mais se voulant plus sécurisé, C s'étant distingué au cours de sa longue histoire par la facilité avec laquelle le·a programmeur·se peut produire des failles de sécurité. Bien d'autres langages sont sur ce créneau, le plus connu (et le seul déjà largement utilisé en production) étant Rust. Mais il y a aussi V, Odin <https://odin-lang.org/>, Vale <https://vale.dev/>...(Une liste a été compilée <https://github.com/robertmuth/awesome-low-level-programming-language

Zig a démarré en 2016 et n'est donc pas un langage si récent que cela. Mais il est toujours officiellement expérimental (l'actuelle version stable est la 0.11, j'ai travaillé avec la 0.12) et le langage continue à évoluer, ce qui rend difficile de l'utiliser pour des projets réels. En outre, sa bibliothèque standard évolue encore davantage que le langage lui-même. Il est ainsi fréquent qu'on trouve avec un moteur de recherche des articles prometteurs...mais dépassés et dont les exemples ne compilent même pas.

Bon, connaissant les lecteurices de ce blog, je pense qu'ielles sont toustes en train de se demander à quoi ressemble "*Hello, world*" en Zig. Donc :

```
const std = @import("std");

pub fn main() !void {
    const public = "Blog";
    std.debug.print("Hello, {s}!\n", .{public});
}
```

Vous noterez qu'il faut importer explicitement la bibliothèque standard, contrairement à la grande majorité des langages de programmation. Une version plus longue de ce premier programme est . En décommentant les lignes qui commencent par deux barres obliques, vous découvrirez en outre que :

— Les variables doivent être utilisées (sinon, le compilateur dira *"error : unused local constant"*),
— on ne rigole pas avec le typage. Par exemple, vous ne pouvez pas ajouter un entier à un pointeur (*"error : incompatible types"*).

Bon, je vous ai suggéré d'essayer mais, pour cela, il faudrait un compilateur. La mise en œuvre actuelle de Zig est en C++ mais une version en Zig existe et est déjà capable de se compiler elle-même. En attendant, on va utiliser le binaire fourni `<https://ziglang.org/download/>` car je suis paresseux :

```
 % zig version
0.12.0-dev.1245+a07f288eb

% zig build-exe hello.zig

% ./hello
Hello, Blog!
```

Vous verrez un message pendant un moment disant que le code est généré par LLVM. Un des avantages de cette méthode est de rendre Zig très portable. Ainsi, les programmes tournent tous sur ma carte RISC-V `<https://www.bortzmeyer.org/star64-first-boot.html>`.

D'autre part, la commande zig joue plusieurs rôles : compilateur et éditeur de liens, bien sûr, mais aussi remplaçant de make et des outils similaires, et gestionnaire de paquetages.

Revenons à des programmes zig et voyons ce qui se passe si le programme plante :

```
const std = @import("std");

pub fn main() void {
  const name = "myfile.txt";
  const f = std.fs.cwd().openFile(name, .{});
  std.debug.print("File opened? {any}\n", .{f});
}
```

Ouvrir un fichier n'est évidemment pas une opération sûre. Le fichier peut être absent, par exemple. Les différents langages de programmation ont des mécanismes très différents pour gérer ces cas. Si vous faites tourner le programme Zig ci-dessus, et que le fichier `myfile.txt` n'existe pas, vous obtiendrez un *"File opened ? error.FileNotFound"*. Si le fichier existe, ce sera *"File opened ? fs.file.File{ .handle = 3, .capable_io_mode = io.ModeOverride__enum_3788.blocking, .intended_io_mode = io.ModeOverride__enum_3788.blocking }"*, une structure de données à travers laquelle on pourra manipuler le fichier ouvert. La fonction de la bibliothèque standard `openFile` peut donc retourner deux types différents, une erreur ou une structure d'accès au fichier. Ce mécanisme est très fréquent en Zig, on peut avoir des unions de type et les fonctions renvoient souvent « soit ce qu'on attend d'elles, soit une erreur ».

Maintenant, imaginons un·e programmeur·se négligent·e qui ignore l'erreur et veut lire le contenu du fichier :

```
  var buffer: [100]u8 = undefined;
  const f = std.fs.cwd().openFile(name, .{});
  const result = f.readAll(&buffer);
  std.debug.print("Result of read is {d} and content is \"{s}\"\n", .{result, buffer[0..result]});
}
```

Ce code ne pourra pas être compilé :

```
% zig build-exe erreurs.zig
erreurs.zig:8:19: error: no field or member function named 'readAll' in 'error{...FileNotFound...BadPathName...}
  const result = f.readAll(&buffer);
                 ~^~~~~~~~
```

En effet, en Zig, on ne peut pas ignorer les erreurs (une faute courante en C). `openFile` ne renvoie pas un fichier (sur lequel on pourrait appliquer `readAll`) mais une union « erreur ou fichier ». Il faut donc faire quelque chose de l'erreur. Zig offre plusieurs moyens pour cela. L'une des plus classiques est de préfixer l'appel de fonction avec `try`. Si la fonction ne renvoie pas d'erreur, `try` permet de ne garder que la valeur retournée, si par contre il y a une erreur, on revient immédiatement de la fonction appelante, en renvoyant l'erreur :

```
  var buffer: [100]u8 = undefined;
  const f = try std.fs.cwd().openFile(name, .{});
  const result = try f.readAll(&buffer);
  std.debug.print("Result of read is {d} and content is \"{s}\"\n", .{result, buffer[0..result]});
```

Mais ça ne compile pas non plus :

```
% zig build-exe erreurs.zig
erreurs.zig:6:13: error: expected type 'void', found 'error{AccessDenied...}'
  const f = try std.fs.cwd().openFile(name, .{});
            ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
erreurs.zig:3:15: note: function cannot return an error
pub fn main() void {
              ^~~~
```

On a dit que `try` renvoyait une erreur en cas de problème. Or, la fonction `main` a été déclarée comme ne renvoyant rien (`void`). Il faut donc la changer pour déclarer qu'elle ne renvoie rien, ou bien une erreur :

```
pub fn main() !void {
```

Le point d'exclamation indiquant l'union du type erreur et du vrai résultat (ici, `void`). Rappelez-vous : on n'a pas le droit de planquer les erreurs sous le tapis. Le programme utilisé est .

Revenons sur le typage Zig :

— Il est strict, on ne peut pas ajouter un entier à un pointeur, par exemple,

— il est entièrement statique (les types ne survivent pas à la compilation),

— résultat (qui ne surprendra pas les programmeurs et programmeuses Rust), arriver à compiler le programme est souvent long et pénible mais, une fois que ça compile, il y a beaucoup moins de bogues qui trainent (ce qui est, entre autre, positif pour la sécurité),
— les types sont aussi des valeurs, on peut les mettre dans des variables, les passer en paramètres à des fonctions (ce qui est utile pour la généricité) mais tout cela est fait à la compilation.

Voyons maintenant les variables optionnelles. Il est courant qu'une variable n'ait pas toujours une valeur, par exemple si la fonction qui lui donnait une valeur échoue. En C, et dans d'autres langages, il est courant de réserver une valeur spéciale pour dire « pas de valeur ». Par exemple 0 pour un entier, la chaine vide pour une chaine de caractères, etc. Le problème de cette approche est que cette valeur se trouve désormais interdite (que faire si l'entier doit vraiment valoir 0 ?) Zig utilise donc le concept de variable optionnelle, variable qui peut avoir une valeur ou pas (un peu come le `MayBe` d'Haskell). On les déclare avec un point d'interrogation :

```
    var i:?u8; // = null;
    var j:?u8; // = 42;
    std.debug.print("Hello, {d}!\n", .{i orelse 0}); // orelse va déballer la valeur (ou mettre 0)

    i = 7;
    std.debug.print("Hello, {any}!\n", .{j});  // Optionnel, donc le format {d} ne serait pas accepté
    std.debug.print("Hello, {d}!\n", .{i orelse 0});
```

Ce code ne sera pas accepté tel quel car, rappelez-vous, les variables doivent être initialisées. Autrement, "*error : expected '=', found ';'*" (message peu clair, il faut bien l'avouer). En donnant une valeur initiale aux deux variables optionnelles, le programme marche. On notera :

— `null` indique que la variable n'a pas de valeur. On peut tester `if (i == null)` si on veut vérifier ce cas.
— `orelse` sert à donner une valeur par défaut lorsqu'on « déballe » la variable et qu'on découvre qu'elle n'a pas de valeur.

Passons maintenant à un sujet chaud en sécurité (car une bonne partie des failles de sécurité dans les programmes écrits en C viennent de là), la gestion de la mémoire. En simplifiant, il y a les langages qui gèrent tout, laissant le programmeur libre de faire autre chose (comme Python) et les langages où la programmeuse doit gérer la mémoire elle-même. La première approche est évidemment plus simple mais la deuxième permet de contrôler exactement la mémoire utilisée, ce qui est indispensable pour l'embarqué et souhaitable pour les gros programmes tournant longtemps (les serveurs Internet, par exemple). Et l'approche manuelle a également l'inconvénient que les humain·es femelles ou mâles font des erreurs (lire de la mémoire non allouée, ou qui a été libérée ou, pire, y écrire).

L'approche de Zig est que la mémoire est allouée via des allocateurs (concept emprunté à C++) et que différents allocateurs ont différentes propriétés. La responsabilité de la personne qui programme est de choisir le bon allocateur. Commençons par un exemple simple :

```
pub fn main() !void {
  var myallo = std.heap.GeneralPurposeAllocator(.{}){};
  const allocator = myallo.allocator();

  var myarray = try allocator.alloc(u8, 10); // Peut échouer par manque de mémoire
  defer allocator.free(myarray);
  std.debug.print("{any}\n", .{myarray}); // Selon l'allocateur utilisé, donnée initialisées ou pas

  for (0..myarray.len) |i| {
myarray[i] = @truncate(i); // @as serait refusé, le type de destination étant trop petit
  }
  std.debug.print("{any}\n", .{myarray});
}
```

Ce programme, :

— crée un allocateur à partir d'une fonction de la bibliothèque standard, `GeneralPurposeAllocator`, allocateur qui privilégie la sécurité sur les performances, et n'est pas optimisé pour une tâche précise (comme son nom l'indique),
— alloue de la mémoire pour dix octets (`u8`), ce qui peut échouer (d'où le `try`),
— enregistre une action de libération de la mémoire (`free`) à exécuter lors de la sortie du bloc (`defer`),
— écrit dans cette mémoire (les fonctions dont le nom commençe par un arobase sont les fonctions pré-définies du langage ; ici, `@truncate` va faire rentrer un compteur de boucle dans un octet, quitte à le tronquer),
— affiche le résultat.

D'une manière générale, Zig essaie d'éviter les allocations implicites de mémoire. Normalement, vous voyez les endroits où la mémoire est allouée.

La puissance du concept d'allocateur se voit lorsqu'on en change. Ainsi, si on veut voir les opérations d'allocation et de désallocation, la bibliothèque standard a un allocateur bavard, `LoggingAllocator`, qui ajoute à un allocateur existant ses fonctions de journalisation. Comme tous les allocateurs ont la même interface, définie par la bibliothèque standard, le remplacement d'un allocateur par un autre est facile :

```
const myallo = std.heap.LoggingAllocator(std.log.Level.debug, std.log.Level.debug);
var myrealallo = myallo.init(std.heap.page_allocator);
const allocator = myrealallo.allocator();

var myarray = try allocator.alloc(u8, 10);
defer allocator.free(myarray);
for (0..myarray.len) |i| {
    myarray[i] = @truncate(i);
}
std.debug.print("{any}\n", .{myarray});

const otherarray = try allocator.alloc(f64, 3);
defer allocator.free(otherarray);
for (0..otherarray.len) |i| {
    otherarray[i] = @floatFromInt(i);
}
std.debug.print("{any}\n", .{otherarray});
```

Ce programme, va afficher les deux allocations et, à la fin du bloc, les deux déallocations :

```
% ./loggingallocator
debug: alloc - success - len: 10, ptr_align: 0
{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
debug: alloc - success - len: 24, ptr_align: 3
{ 0.0e+00, 1.0e+00, 2.0e+00 }
debug: free - len: 24
debug: free - len: 10
```

Que se passe-t-il si le·a programmeur·se se trompe et, par exemple, utilise de la mémoire qui a été désallouée ? Le résultat dépend de l'allocateur, et des sécurités qu'il fournit (d'où l'importance du choix de l'allocateur). Par exemple, ce code () :

```
    var myallo = std.heap.GeneralPurposeAllocator(.{}){};
    const allocator = myallo.allocator();
    var myarray = try allocator.alloc(u8, 10);
    for (0..myarray.len) |i| {
        myarray[i] = @truncate(i);
    }
    allocator.free(myarray);
    std.debug.print("Use after free: {any}\n", .{myarray});
```

va, avec cet allocateur, provoquer un plantage puisqu'on utilise de la mémoire qu'on vient de désallouer :

```
% ./memoryerrors
Use after free: { Segmentation fault at address 0x7f55f88e0000
```

Mais si on utilise à la place l'allocateur de la libc (plus rapide mais, comme vous le savez, beaucoup moins sûr), aucune erreur ne se produit (ce qui, en dépit des apparences, est un problème) :

```
    const allocator = std.heap.c_allocator;
    var myarray = try allocator.alloc(u8, 10);
    for (0..myarray.len) |i| {
        myarray[i] = @truncate(i);
    }
    allocator.free(myarray);
    std.debug.print("Use after free: {any}\n", .{myarray});
    var myotherarray = try allocator.alloc(u8, 10);
    allocator.free(myotherarray);
    allocator.free(myotherarray);
    std.debug.print("Double free: {any}\n", .{myarray});
```

On doit le compiler en liant avec la libc (sinon, *"C allocator is only available when linking against libc"*) :

```
% zig build-exe memoryerrors.zig -lc

% ./memoryerrors
Use after free: { 212, 33, 0, 0, 0, 0, 0, 0, 59, 175 }
Double free: { 116, 99, 29, 2, 0, 0, 0, 0, 59, 175 }
```

En Zig, les bibliothèques qui ont besoin d'allouer de la mémoire demandent qu'on leur fournisse un allocateur. Si on prend comme exemple `std.fmt.allocPrint`, qui formate une chaine de caractères, sa documentation précise qu'elle attend comme premier paramètre une variable de type `std:mem.Allocator`. L'allocateur à usage généraliste convient donc :

```
    const a = 2;
    const b = 3;
    var myallo = std.heap.GeneralPurposeAllocator(.{}){};
    const allocator = myallo.allocator();

    const mystring = try std.fmt.allocPrint(
        allocator,
        "{d} + {d} = {d}",
        .{ a, b, a + b },
    );
    defer allocator.free(mystring);
    std.debug.print("{s}\n", .{mystring});
```

(Source complet en .)

Zig permet évidemment d'utiliser des bibliothèques existantes, et d'écrire les siennes. Voyons un exemple avec le DNS. On va utiliser la bibliothèque zig-dns <https://github.com/dantecatalfamo/zig-dns>. Elle-même dépend de zig-network <https://github.com/MasterQ32/zig-network>, pour envoyer et recevoir des paquets. On les télécharge :

```
git clone https://github.com/MasterQ32/zig-network.git
git clone https://github.com/dantecatalfamo/zig-dns.git
```

Puis on écrit un programme qui les utilise, pour obtenir l'adresse IP de `www.afnic.fr` :

```
// On importe les deux bibliothèques
const network = @import("zig-network/network.zig");
const dns = @import("zig-dns/src/dns.zig");

// Beaucoup de bibliothèques ont une fonction init, pour... les initialiser.
try network.init();

// On crée la prise réseau et on se ≪ connecte ≫ à son résolveur DNS, ici
// celui de DNS.sb
const sock = try network.connectToHost(allocator, "2a09::", 53, .udp);
const writer = sock.writer();

// La requête
const message = try dns.createQuery(allocator, "www.afnic.fr", .AAAA);
var message_bytes = try message.to_bytes(allocator);
try writer.writeAll(message_bytes);

// On lit la réponse
var recv = [_]u8{0} ** 1024;
const recv_size = try sock.receive(&recv);
const response = try dns.Message.from_bytes(allocator, recv[0..recv_size]);
std.debug.print("Response:\n{any}\n", .{response});
```

Compilé et exécuté, ce programme nous donne :

```
% ./use-zig-dns
...
Response:
Message {
  Header {
    ID: 1
    Response: true
...
 Answers {
    Resource Record {
      Name: www.afnic.fr.
      Type: AAAA
      Class: IN
      TTL: 581
      Resource Data Length: 16
      Resource Data: 2a00:0e00:0000:0005:0000:0000:0000:0002
    }
```

(L'adresse IP de `www.afnic.fr` est `2a00:e00:0:5::2`, même si l'affichage par défaut ne met pas en œuvre la compression du RFC 5952 [1].) Le code complet est en . Notez que, comme indiqué au début, Zig est encore expérimental. Les bibliothèques sont peu nombreuses, pas forcément très documentées, pas toujours bien testées, et parfois ne compilent pas avec les dernières versions du langage.

Zig n'a pas de macros. Les macros qui permettent de changer la syntaxe (comme le préprocesseur C) sont jugées trop dangereuses car rendant difficile la lecture d'un programme. Et celles qui ne changent pas la syntaxe sont moins nécessaires en Zig où on peut faire bien des choses à la compilation. Mais si vous voulez des exemples de tâches qu'on accomplirait avec une macro en C, regardez cette question que j'ai posée sur un forum <`https://ziggit.dev/t/outreach-looking-for-a-simple-example-of-using-`
>, avec plusieurs très bonnes réponses.

Zig est fortement typé et un tel niveau de vérification peut poser des problèmes lorsqu'on veut, par exemple, mettre en œuvre des structures de données génériques. Par exemple, une file d'attente FIFO de variables d'un type quelconque. Zig résout en général ce problème avec l'utilisation de types pour paramétrer un programme. Ces types comme variables doivent pouvoir être évalués à la compilation, et sont donc souvent marqués avec le mot-clé `comptime` ("*compilation time*") :

```
pub fn Queue(comptime Child: type) type {
    return struct {
        const This = @This();
        const Node = struct {
            data: Child,
            next: ?*Node,
        };
```

Ici, la file d'attente est programmée sous forme d'un enregistrement (`struct`), qui comprend des données d'un type `Child` quelconque (mais connu à la compilation). Notez qu'une file est créée en appelant la fonction `Queue` qui renvoie un enregistrement qui comprend, entre autres, les fonctions qui opéreront sur les files d'attente (permettant de faire partiellement de la programmation objet). Le code complet est en (copié du site officiel <`https://ziglang.org/learn/samples/#generic-types`>).

Zig se veut un concurrent de C mais, naturellement, ce n'est pas demain la veille que tout le code existant en C sera recodé en Zig, même dans les hypothèses les plus optimistes. Il est donc nécessaire de pouvoir utiliser les bibliothèques existantes écrites en C. C'est justement un des points forts de Zig : il peut importer facilement des déclarations C et donc permettre l'utilisation des bibliothèques correspondantes. Voyons tout de suite un exemple. Supposons que nous avons un programme en C qui calcule le PGCD de deux nombres et que nous n'avons pas l'intention de recoder (dans ce cas précis, il est suffisamment trivial pour être recodé, je sais) :

```
unsigned int
pgcd(unsigned int l, unsigned int r)
{
    while (l != r) {
        if (l > r) {
            l = l - r;
        } else {
            r = r - l;
        }
    }
    return l;
}
```

1. Pour voir le RFC de numéro NNN, `https://www.ietf.org/rfc/rfcNNN.txt`, par exemple `https://www.ietf.org/rfc/rfc5952.txt`

Zig peut importer ce programme (la commande `zig` inclut clang pour analyser le C) :

```
const std = @import("std");
const c = @cImport(@cInclude("pgcd.c")); // No need to have a .h

pub fn main() !void {
    ...
    std.debug.print("PGCD({d},{d}) = {d}\n", .{ l, r, c.pgcd(l, r) });
}
```

On notera qu'il n'y a pas besoin de déclarations C dans un fichier `.h` (mais, évidemment, si vous avez un `.h`, vous pouvez l'utiliser). Le programme se compile simplement (et le `pgcd.c` est compilé automatiquement grâce à clang) :

```
% zig build-exe find-pgcd.zig  -I.

% ./find-pgcd 9 12
PGCD(9,12) = 3
```

Pour compiler, au lieu de lancer `zig build-exe`, on aurait pu créer un fichier `build.zig` contenant les instructions nécessaires et, après un simple `zig build` fait tout ce qu'il faut (ici, l'application était suffisamment simple pour que ce ne soit pas nécessaire). Un exemple d'un tel fichier est (et le code C est et le programme principal).

Prenons maintenant un exemple d'une vraie bibliothèque C, plus complexe. On va essayer avec la libidn <https://www.gnu.org/software/libidn/>, qui permet de gérer des noms de domaines internationalisés (noms en Unicode, cf. RFC 5891). On va essayer de se servir de la fonction `idn2_to_ascii_8z`, qui convertit un nom en Unicode vers la forme Punycode (RFC 5892). D'abord, on va importer l'en-tête fourni par cette bibliothèque (au passage, n'oubliez pas d'installer les fichiers de développement, sur Debian, c'est le paquetage `libidn2-dev`) :

```
const idn2 = @cImport(@cInclude("idn2.h"));
```

Puis on crée les variables nécessaires à la fonction qu'on va utiliser :

```
var origin: [*c]u8 = args[1];
var destination: [*:0]u8 = undefined;
```

Le type `*c` indique une chaine de caractères sous forme d'un pointeur du langage C, quant à `[*:0]`, c'est une chaine terminée par l'octet nul, justement ce que la fonction C va nous renvoyer. On peut donc appeler la fonction :

```
const result: c_int = idn2.idn2_to_ascii_8z(orig, @ptrCast(&dest),
                                            idn2.IDN2_NONTRANSITIONAL);
```

Ce code peut ensuite être compilé et exécuté :

```
% ./use-libidn café.fr
Punycode(café.fr) = xn--caf-dma.fr (return code is 0)
```

Comment est-ce qe j'ai trouvé quels paramètres exacts passer à cette fonction ? La documentation <https://www.gnu.org/software/libidn/manual/libidn.html#Simplified-ToASCII-Interface> conçue pour les programmeurs C suffit souvent mais, si ce n'est pas le cas, si on connait C meux que Zig, et qu'on a un exemple de code qui marche en C, une technique bien pratique est de demander à Zig de traduire cet exemple C en Zig :

```
% zig translate-c example.c -lc > example.zig
```

Et l'examen du `example.zig` produit vous donnera beaucoup d'informations.

Notez que je n'ai pas alloué de mémoire pour la chaine de caractères de destination, la documentation (pas très claire, je suis d'accord) disant que la libidn le fait.

Autre problème, le code Zig doit être lié à la libc explicitement (merci à Ian Johnson <https://ziggit.dev/u/ianprime0509/> pour l'explication). Si vous ne le faites pas, vous n'aurez pas de message d'erreur mais (dépendant de votre plate-forme), une "*segmentation fault*" à l'adresse...zéro :

```
% zig build-exe use-libidn.zig -lidn2
% ./use-libidn café.fr
Segmentation fault at address 0x0
???:?:?: 0x0 in ??? (???)
zsh: IOT instruction  ./use-libidn café.fr

% zig build-exe use-libidn.zig -lidn2 -lc
% ./use-libidn café.fr
Punycode(café.fr) = xn--caf-dma.fr (return code is 0)
```

Si vous utilisez un `build.zig`, n'oubliez donc pas le `exe.linkLibC();`.

Si vous voulez essayer vous-même, voici et (à renommer en `build.zig` avant de faire un `zig build`). Si vous voulez d'autres exemples, vous pouvez lire l'article de Michael Lynch <https://mtlynch.io/notes/zig-call-c-simple/>.

Une des forces de Zig est l'inclusion dans le langage d'un mécanisme de tests <https://ziglang.org/documentation/master/std/#A;std:testing> permettant de s'assurer du caractère correct d'un programme, et d'éviter les régressions lors de modifications ultérieures. Pour reprendre l'exemple du calcul du PGCD plus haut, on peut ajouter ces tests à la fin du fichier :

```
const expect = @import("std").testing.expect;

fn expectidentical(i: u32) !void {
    try expect(c.pgcd(i, i) == i);
}

test "identical" {
    try expectidentical(1);
    try expectidentical(7);
    try expectidentical(18);
}

test "primes" {
    try expect(c.pgcd(4, 13) == 1);
}

test "pgcdexists" {
    try expect(c.pgcd(15, 35) == 5);
}

test "pgcdlower" {
    try expect(c.pgcd(15, 5) == 5);
}
```

Et vérifier que tout va bien :

```
%   zig test find-pgcd.zig -I.
All 4 tests passed.
```

Il n'y a pas actuellement de « vrai » programme écrit en Zig, « vrai » au sens où il serait utilisé par des gens qui ne connaissent pas le langage de programmation et s'en moquent. On lira toutefois un intéressant article sur le développement de Bun <https://changelog.com/jsparty/295>.

Si ce court article vous a donné envie d'approfondir Zig, voici quelques ressources en ligne :

— Le site officiel <https://ziglang.org/>,
— Plus pédagogiques et progressifs, Ziglearn <https://ziglearn.org/> ou les géniaux Ziglings <https://ziglings.org/>, où on apprend en corrigeant des programmes erronés (avec des indications, rassurez-vous),
— Côté forums, on a Zigg-it <https://ziggit.dev/>, Zig NEWS <https://zig.news/> ou bien sûr StackOverflow <https://stackoverflow.com/questions/tagged/zig> (mais où les ziggistes semblent moins actifs),
— Et si vous hésitez entre Zig et d'autres langages, il y a une bonne comparaison avec les langages concurrents <https://ziglang.org/learn/why_zig_rust_d_cpp/>.