Langage C avancé

CSC_3INM2_TA
ENSTA Paris - TC 1ère année

François Pessaux

U2IS

2024-2025

prenom.nom@ensta-paris.fr

38 ▶ Pointeurs sur fonctions et interruptions

Préambule

Avant de commencer. . .

Moi == François Pessaux
U2IS, bureau R223

- Ne pas hésiter à poser des questions, même en cours.
- Posez votre question que vous jugez « bête » : d'autres ici ont la même.
- Ne pas hésiter à dire « *M'sieur, vous vous z'êtes pas trompé sur xyz ?* »
- Mon bureau et mon mail sont facilement accessibles.
- Ne pas hésiter à venir me voir, me parler. Si si !
- Terminez et conservez vos exercices, vos programmes.

Pourquoi dois-je subir des cours d'informatique ?

- Numérique[1] présent dans tous les domaines, tous les métiers.
- Ingénieurs de formation généraliste.
- Pour vos cours futurs (vue pragmatique à court terme).
- Pour votre future carrière :
  - Simulation numérique (mécanique, finance, etc.) ? $\Rightarrow$ info + ...
  - Systèmes embarqués ? $\Rightarrow$ info + ...
  - Data-science ? $\Rightarrow$ info + ...
  - Recherche opérationnelle ? $\Rightarrow$ info + ...
  - Cyber-sécurité ? $\Rightarrow$ info + ...
  - Chef de projet au sens large ? $\Rightarrow$ info + ...
  - Etc.

1. et pas « digital », qui est relatif au doigt !

Avant / Maintenant

- Vous avez déjà vu :
  - Des langages de programmation : `Python`, `OCaml`, `C`.
  - Utilisation de bibliothèques.
  - Écriture des programmes pour des problèmes relativement modestes.
- Buts de ce cours :
  - Réviser les notions de `C` déjà vues en prépa.
  - Approfondir certains traits de `C`.
  - Comprendre l'intérêt d'un langage (si) rudimentaire mais efficace.
  - Comprendre l'intérêt de certaines constructions plus avancées.
  - Pratiquer `C` sur des programmes restant assez modestes.

L'informatique en première année à l'

ENSTA

IP PARIS

MO101

- Configuration de votre machine personnelle pour les cours
- Initiation à Unix et à l'interprète de ligne de commande
- Initiation à LaTeX
  - Éditer un texte
  - Utiliser le terminal

IN102

Prérequis

- Apprendre un nouveau langage
  - syntaxe
  - sémantique
- Comprendre la gestion de la mémoire
  - Savoir programmer en C

IN101

Prérequis

- Apprendre à analyser la spécification d'un programme
- Apprendre à concevoir des algorithmes
- Acquérir une démarche systématique et scientifique
  - d'analyse
  - de formalisation
  - de conception
  - d'implantation
- Pratiquer C
  - Concevoir et implanter un algorithme
  - Être soucieux de sa complexité et de sa correction

IN103

Prérequis

- Étudier des structures de données fondamentales
- Étudier des opérations sur ces structures de données
- Concevoir des algorithmes et programmes plus complexes
- Pratiquer C
  - Choisir et implanter des structures de données

IN104

Prérequis

- Réaliser un développement informatique conséquent
- Travailler en autonomie et en équipe
- Utiliser une méthodologie et des outils de travail collaboratifs

2A maths appliquées

Programmation scientifique en C++
Projet de simulation numérique
Bases de données
Optimisation Différentiable: théorie et algorithmes
...

Majeure	Métiers accessibles
Mathématiques appliquées	Métiers en Recherche (§1) Ingénieur recherche opérationnelle/optimisation (§4) Ingénieur Gestion des risques financiers (§4) Ingénieur modélisation et simulation (§4) Data Scientist (§4)

2A info

POO et Génie Logiciel
Information et réseaux
Programmation parallèle
Applications mobiles
Bases de données
...

Majeure	Métiers accessibles
Informatique	Métiers en Recherche (§1) Métiers en Etudes et conception (§2) Ingénieur en Intelligence artificielle (§3) Ingénieur cybersécurité / Architecte système d'information (§3) Ingénieur sureté de fonctionnement (§5)

2A mécanique

Introduction aux réseaux
Programmation orientée objet et embarquée
Reconnaissance d'images
...

Majeure	Métiers accessibles
Mécanique	Métiers en Recherche (§1) Ingénieur système mécanique complexe / ingénieur d'études (§2) Ingénieur calculs (§2) Ingénieur Maitrise des risques industriels (§5)

Attention !

Pratiquez !

Juste réviser 3 jours à l'avance ne suffira pas si vous ne vous êtes pas exercés (idem pour IN101, IN103, IN104) !

(Mais je pense que je prêche des convaincus)

Modalités d'évaluation des connaissances

- Informations sur IN102 « standard » : `https://perso.ensta-paris.fr/~frehse/in102web/`
  - Poly : `https://perso.ensta-paris.fr/~frehse/in102web/in102-poly.pdf`
  - Notebooks : `https://perso.ensta-paris.fr/~frehse/in102web/notebooks.html`

- Même examen final que le cours IN102 « standard ».
- Chaque séance, temps accordé pour faire le QCM.
- Pas de « TDs validés » (format non adapté ici) mais équivalent en bonus de participation.
- Résumé :
  - 20% de QCM,
  - 20% de participation en TDs,
  - 60% d'examen final.

Quelques rappels de `C`

C pourquoi ?

`[MODE publicité=ON]`

- Très utilisé dans l'industrie malgré certains défauts/difficultés.
- « *Souple* » (programmation haut et bas niveau).
- Disponible quelle que soit (à 99.9999%) l'architecture.
- Seul moyen « *simple* » de piloter du matériel.
- Excellentes performances (sauf sur un algo pourri bien sûr).
- Bibliothèques et outils de développement en quantité.

`[MODE publicité=OFF]`

`[MODE inconvénient=ON]`

- Gestion de la mémoire explicite.
- Structures de données de base rudimentaires.
- Fait apparaître des détails de compilation du langage.

`[MODE inconvénient=OFF]`

Le sempiternel et incontournable premier programme en C

```
#include <stdio.h>

int main () {
  printf ("Eat at Joe's\n") ;
  return 0 ;
}
```

- La plus simple manière d'afficher une simple chaîne de caractères.
- `/* ... */` : Commentaires.
  - Ignorés par le langage.
  - **Mais** pas par les lecteurs !
  - Servent à la documentation du code.
  - **Documentez** vos programmes !
- Fonction `main` : **Le** point d'entrée du programme.
- `main` retourne un entier de statut (OK / erreur).

Édition du code source

- Utilisation de `Visual Studio Code`.
- Montage de votre *homedir* avec `sshfs` via `salle.ensta.fr`
  - `sshfs` *`votre identifiant`*`@salle.ensta.fr:`
    `~/quelque-part/remote-files`
- Compil. / éxec. sur le serveur distant `info1.ensta.fr`
  - `ssh -X` *`votre identifiant`*`@info1.ensta.fr`
- Fichiers stockés sur le serveur et non sur votre machine.
- Possibilité de rapatrier les fichiers sur votre machine.
- Procédure vue en MO101.
- $\Rightarrow$ Homogénéité de l'environnement pour tout le monde.

Au boulot…

- Montez votre *homedir* avec `sshfs` via `salle.ensta.fr`.
- Connectez-vous à `info1.ensta.fr`.
- Créez un nouveau fichier `C`.
- Écrivez le programme précédent.

Compiler/exécuter

- Compilation : `gcc -Wall -Werror -Wfatal-errors joe.c`
- Exécution : `./a.out`

- Compilation :
  `gcc -Wall -Werror -Wfatal-errors joe.c -o joe.x`
- Exécution : `./joe.x`

- Surtout pas : ~~`gcc joe.c -o joe.C`~~
  - ⇒ détruirait votre fichier source.
- `-Wall` indispensable pour vérifier l'absence de *warnings*.
- `-Werror` indispensable pour considérer les *warnings* comme erreurs.
- `-Wfatal-errors` indispensable pour arrêter la compilation à la première erreur.

Au boulot. . .

Compilez votre code source.

Exécutez-le.

Le compilateur gcc

- Compilateur : programme transformant un source en exécutable.
- Dispose de nombreuses options dont :
  - `-W` : Activer des warnings (`-Wall` : tous, plus que recommandé).
  - `-g` : Activer les infos de débug.
  - `-I` : Ajouter des répertoires où trouver les entêtes (`#include`).
  - `-c` : Compile sans « *linker* » (lorsque plusieurs fichiers sources).
  - `-L` : Transfère des options au « *linker* »
  - `-O0 -O1 -O2 -O3` : Optimise pas, peu, plus, beaucoup (trop).
  - `-l` : Lier une bibliothèque (Ex : `-lm` pour la bibliothèque maths).
- Si tout s'est bien passé, le compilateur n'affiche rien.
- S'il affiche une erreur, l'exécutable n'est pas créé.
- S'il affiche un/des warning/s, l'exécutable est créé, mais risque de ne pas fonctionner correctement.

- **Types et opérations de base**

Types de base

- Scalaires

Nom	*Taille(octets)*	*Description*	*Exemple*
`char`	1	caractère ASCII	`'a'`, `'n'`
`short`	2	entier court signé	-32768, 32767
`int`	4	entier signé	10, -1
`long (int)`	8	entier long signé	
`float`	4	flottant simple précision	3.14159
`double`	8	flottant double précision	

- Avec `#include <stdbool.h>`

Nom	*Taille(octets)*	*Description*	*Exemple*
`bool`	1	booléen	`false`, `true`

- Modificateur `unsigned` pour les entiers non signés.
- On reviendra plus tard sur ces types...

Opérations de base sur les types de base

- Arithmétique : entre entiers et/ou flottants.
  - +, -, /, * avec leur sens « habituel », % (modulo)
  - Ex : `4 - (5 * 7)`
- Relationnel (« tests ») : entre expressions de même type.
  - Rem : conversions implicites entre scalaires (entiers, flottants, caractères et booléens).
  - `==` (égalité), `!=` (inégalité), <, >, <=, >=.
  - Ex : `y <= 5`   `(6 * 4)< x`
- Logique : entre booléens (donc implicitement, entiers).
  - `&&` (et), `||` (ou), `!` (négation).
  - Ex : `(y <= 5)&& ((6 * 4)< x)`
  - `&&` et `||` sont « *paresseux* ».
- Binaire (« bit-à-bit ») : entre entiers / caractères.
  - Fonction logique opérant sur chacun des bits de la représentation.
  - ~ (négation), ^ (ou exclusif), & (et), | (ou).
  - Ex : `(~x & y)^ 0xFE`
  - On y reviendra plus tard...

Variables locales, variables globales

- Variables déclarées avec leur type.
  - `unsigned int age = 18 ;`
- On peut déclarer des variables locales dans une fonction.
- Plus généralement : déclaration de variables locales dans des blocs ({ ... }).
- Corps de la fonction ≡ bloc.
- ⇒ Vivantes seulement jusqu'à la fin de la fonction / du bloc.
- Variables globales (définies hors de fonctions) restent accessibles dans la fonction.

```
int glob = 5 ;         /* Variable globale. */

int f (int x) {
  int i = x * 2 ;      /* Variable locale à la fonction f. */
  {
    int j = i + 5 ;    /* Variable locale au bloc interne à f. */
    i = i + j ;
  }                    /* Fin du bloc: j n'est plus accessible. */
  return i + 3 + glob ;
}
```

L'affectation

- Dénotée par le signe = (à ne pas confondre avec ==).
- Raccourcis : _ ♪= _

Syntaxe	Signification
`i += 3`	`i = i + 3`
`i -= 3`	`i = i - 3`
`i *= 3`	`i = i * 3`
`i /= 3`	`i = i / 3`
`i %= 3`	`i = i % 3`
`i |= 3`	`i = i | 3`
`i &= 3`	`i = i & 3`
`i ^= 3`	`i = i ^ 3`
`i++`, `++i`	`i = i + 1`
`i--`, `--i`	`i = i - 1`

Au boulot...

- Quelle est la différence entre `++i` et `i++` ?
- Si vous ne savez pas, examinez le résultat du programme suivant :

```
#include <stdio.h>

int main () {
  int x = 0, y, z ;
  y = x++ ;
  z = ++x ;
  printf ("x: %d, y: %d, z: %d\n", x, y, z) ;
  return 0 ;
}
```

- Concluez.

- **Structures de contrôle**

Conditionnelle (instruction / expression)

- `if` **(** *expression* **)** { *instruction*(*s*) `;` } :
  - *instruction*(*s*) exécutée(s) si *expression* renvoie $\neq 0$

  instruction(s) exécutée(s) si expression renvoie $\neq 0$.
- `if` **(** *expression* **)** { $instruction_1(s)$ `;` } `else` { $instruction_2(s)$ `;` } :
  - $instruction_1(s)$ exécutée(s) si *expression* renvoie $\neq 0$
  - sinon, $instruction_2(s)$ exécutée(s).
- Condition toujours entre parenthèses !
- Conditionnelle ternaire : $expression_1$ `?` $expression_2$ `:` $expression_3$
  ▶ Est une expression $\equiv$ le `if` d'`OCaml`.

Boucles

- `while` (*expression*) { *instruction*(*s*) }
- `do` { *instruction*(*s*) } `while` (*expression*) ;
- Condition toujours entre parenthèses !
- `for` ($instruction_1$; *expression* ; $instruction_2$) { $instruction_3(s)$ }
  - ≡ $instruction_1$;
    `while` (*expression*) {
      $instruction_3(s)$;
      $instruction_2$;
    }

La boucle for dégénérée

- Possibilité d'omettre des parties de la construction.
- Initialisation, condition, post-traitement, voire corps !
- Poussé à l'extrême : formes étranges voire illisibles. À éviter !

```
i = 0 ; j = 0 ;
for (/**/; i < 10; i++) {
  j += i ;
}
```

```
i = 0 ; j = 0 ;
for (i = 0; i < 10;/**/) {
  j += i ;
  i++ ;
}
```

```
i = 0 ; j = 0 ;
for (/**/;i < 10;/**/) {
  j += i ;
  i++ ;
}
```

```
i = 0 ; j = 0 ;
for (/**/;/**/;/**/) {
  j += i ;
  if (i == 9) break ;
  i++ ;
}
```

- On reviendra sur le `break` plus tard.

Au boulot. . .

- Afficher `nb_ligne = 5` de `n = 20` caractères, où les étoiles s'alternent avec des espaces à partir de la colonne numéro `k = 9`. Les étoiles doivent s'alterner aussi sur les lignes.
- Doit fonctionner pour d'autres valeurs de `nb_lignes`, `n` et `k`.
- Sortie :

```
␣␣␣␣␣␣␣␣␣***********↵
␣␣␣␣␣␣␣␣␣␣*********␣↵
␣␣␣␣␣␣␣␣␣***********↵
␣␣␣␣␣␣␣␣␣␣*********␣↵
␣␣␣␣␣␣␣␣␣***********↵
```

- **Fonctions**

Fonction

- type-retour nom-fonction (type-argument, . . .) { corps }
  - `int min (int x, int y){ return x < y ? x : y ;}`
- Fonction « ne retournant rien » : retour `void`.
  - `void prdouble (int x){ printf ("%d\n", x * 2); }`
  - `return` « sans rien » optionnel.
  - Peut être nécessaire si plusieurs points de terminaison.
- Fonctions implicitement récursives.
- Récursion mutuelle : nécessite présence de prototypes (plus tard).

- **Entrées / sorties / arguments de ligne de commande**

Les formats de `printf` (1/2)

- Nécessite : `#include <stdio.h>`
- Format = chaîne de caractères contenant (ou pas) des séquences %♪.

`%d`	un `int`
`%ld`	un `long int` en décimal
`%u`	un `unsigned int` en décimal
`%x`	un `int` en hexadécimal
`%f`	un `float`
`%lf`	un `double`
`%e`	un `double` en notation scientifique
`%.7lf`	un `double` avec 7 chiffres après la virgule
`%07d`	un `int` en décimal sur 7 digits (remplissage frontal avec des 0)
`% 7d`	un `int` en décimal sur 7 digits (remplissage frontal avec des ' ')
`%c`	un `char` (comme caractère ASCII, pas comme entier)

Les formats de printf (1/2)

```c
#include <stdio.h>

int main () {
  printf ("%d\n", 42) ;
  printf ("%ld\n", 42) ;
  printf ("%u\n", 42) ;
  printf ("%x\n", 42) ;
  printf ("%f\n", 42.0) ;
  printf ("%e\n", 42.0) ;
  printf ("%.7f\n", 42.0) ;
  printf ("%07d\n", 42) ;
  printf ("% 7d\n", 42) ;
  printf ("%c\n", 42) ;
  return 0 ;
}
```

```
42
42
42
2a
42.000000
4.200000e+01
42.0000000
0000042
     42
*
```

Acquérir des entrées : `scanf`

- Nécessite : `#include <stdio.h>`
- Lecture au clavier selon le format spécifié (séquences %♪).
- Passage par adresse des variables réceptacles.
- Si saisie pas en accord avec ce qu'attend le format : imprévisible.
- Format avec uniquement des % (pas de « *message* »).

input.c

```c
#include <stdio.h>

int main () {
  int i, j ;
  scanf ("%d %d", &i, &j) ;
  printf ("i=%d, j=%d\n", i, j);
  return 0 ;
}
```

```
$ gcc input.c -o input
$ ./input
45 67
i=45, j=67
$ ./input
5
FHG
i=5, j=0
$ ./input
DFG 6
i=0, j=0
```

Acquérir des entrées : arguments de ligne de commande

- Forme alternative du main
  - `int main (int argc, char *argv[])`
- `argc` : nombre d'entrées dans `argv`.
- `argv` : tableau de chaînes de caractères.
- 1er élément : nom + chemin de l'exécutable.
- Éléments suivants : arguments effectifs (chaînes de caractères).
- Ex : `$ ./bin/myprog 1 hop 4`

  ⇒ `argv` =

0	1	2	3
"./bin/myprog"	"1"	*"hop"*	"4"

Transformer les chaînes de la ligne de commande (bof)

- argv « *contient* » uniquement des chaînes de caractères.
- Fonctions à disposition (requièrent #include <stdlib.h>) :
  - atoi string → int
  - atol : string → long int
  - atoll : string → long long int
  - atof : string → float
- Bof, bof : ne permet pas de détecter les erreurs :-/

```
#include <stdio.h>
#include <stdlib.h>  /* Pour accéder à atoXXX */
int main (int argc, char *argv[]) {
  int i = atoi (argv[1]) ;
  long l = atol (argv[2]) ;
  long long ll = atoll (argv[3]) ;
  float f = atof (argv[4]) ;
  printf ("i: %d, l: %ld, ll: %lld, f: %f\n", i, l, ll, f) ;
  return 0 ;
}
```

```
$ gcc -Wall string-to-nums.c
$ ./a.out 1 2 4559924234322424 4.5
i: 1, l: 2, ll: 4559924234322424, f: 4.500000
```

Transformer les chaînes de la ligne de commande (mieux)

- `long strtol (char *str, char **endptr, int base)`
  - string → long int
  - Permet de spécifier la base de conversion ($\in [2, 36]$).
  - Si erreur, retourne 0 et positionne `errno` à $\neq 0$
    - `EINVAL` ou `ERANGE`
  - Si `endptr != NULL`, contient adresse 1er octet invalide
- Et consorts...
  - `strtoll` : string → long long int
  - `strtoul` : string → unsigned long int
  - `strtoull` : string → unsigned long long int
  - `strtof` : string → float
  - `strtod` : string → double
  - `strtold` : string → long double

Au boulot...

Écrivez un programme qui prend en argument (ligne de commande) des nombres entiers et qui affiche le plus grand de ces nombres.

Au boulot...

Écrivez un programme qui affiche les arguments de ligne de commande qu'il a reçus.

Contrainte : votre programme **ne devra pas** utiliser de boucles et ne devra comporter que **1 seule** fonction (qui elle, a le droit d'appeler ce qu'elle veut). De même, vous n'avez évidemment pas le droit à l'instruction diabolique `goto` (que nous verrons plus tard avec toutes les précautions qui s'imposent).

Si cela peut vous simplifier, vous pouvez afficher les arguments en ordre inverse de celui sur la ligne de commande.

- **Types de données composites**

Types de données composites

- `struct` : structure regroupant plusieurs variables de types potentiellement différents dans un même type.
  - Définition type : `struct point { double x; double y ; };`
  - Définition variable : `struct point p = { 1.5, 3.6 } ;`
  - Accès avec la notation pointée : `p.x = 2.0 ;`
- `enum` : regroupement de plusieurs constantes symboliques (entières) dans un même type.
  - Définition type : `enum color { RED, BLUE, GREEN };`
  - Définition variable : `enum color col = BLUE ;`
- `type t[SIZE]` : tableau statique de `SIZE` éléments.
  - Éléments tous du même type `type`.
  - Indices $\in [0, \mathtt{SIZE} - 1]$.
- Chaînes de caractères : tableaux de caractères terminés par le caractère `'\0'`.

Au boulot...

Donnez une représentation avec les structures de données que vous connaissez pour une position GPS (par exemple 48°42'39.1"N 2°13'09.4"E) composée des informations suivantes :

- orientation de latitude (Nord ou Sud),
- degrés et minutes de latitude (entiers),
- secondes et fractions de latitude (flottants),
- orientation de longitude (Est ou Ouest),
- etc. comme pour la latitude mais pour la longitude...

- **Pointeurs**

Pointeurs

- Déclaration grâce au caractère * à droite du type lors de la déclaration de variable :
  - `int *a ;` a est un pointeur sur un entier.
  - `char *b ;` b est un pointeur sur un caractère.
  - `int *c[2] ;` c est un tableau de pointeurs sur des entiers.
  - `struct point *p ;` p est un pointeur sur une structure de type `struct point`.
- Contient une adresse.
- Manipulation :
  - `a` adresse d'entier pointé (dont l'adresse est dans `a`).
  - `*a` valeur stockée à l'adresse contenue dans `a`.
  - `int x ; a = &x ;` & donne l'adresse de la variable `x`.
  - `p->x = 2.0 ;` identique à `(*p).x`, raccourci pour déréfencer un pointeur sur une structure.

Au boulot...

Soit le programme suivant :

```
{
  int a, b, *c, *d, **e ;
  a = 3 ; b = 5 ;
  c = &b ; d = &a ;
  *c = 4 ;
  *d = (*c) + 3 ;
  e = &c ;
  **e = *d ;
  *e = &a ;
}
```

Que valent `a` et `b` à la fin de son exécution ?

Au boulot…

Qui a dit que l'opérateur « puissance » `**` n'existait pas en `C` ?

```
#include <stdio.h>

int main () {
  printf ("%d\n", 50 ** "2") ;
  return 0 ;
}
```

- Testez.
- Expliquez.

- **Allocation mémoire**

Allocation mémoire

- Allocation statique :
  - Si la taille est connue à la compilation et modeste.
  - Si le tableau n'est pas valeur de retour d'une fonction.

```
#define SIZE (20)
char name[SIZE] ;
```

- Allocation dynamique : dans les autres situations.
  - Allocation mémoire : `malloc` (taille en octets).
  - Libération mémoire : `free` (pas de taille).

```
double *t = malloc (10 * sizeof (double)) ;
if (t == NULL) { /* Erreur à gérer. */ }
... /* Plus tard au bon moment. */
free (t) ;
```

  - Nécessite l'opérateur `sizeof` (on y reviendra plus tard).

Allocation dynamique et erreurs courantes

- Ne pas initialiser un pointeur (i.e. ne pas allouer via `malloc`).
- Accéder en dehors d'une zone allouée (débordement).
- Faire `free` dans une boucle (désallocations multiples de la zone).
- Faire `free` avec une adresse non allouée dynamiquement.
- Faire `free` avec une adresse qui n'est pas le début de la zone.
- « Perdre » l'adresse d'une zone allouée :

```
int *p = malloc (...) ;
...   /* On ne mémorise pas la valeur de p ailleurs. */
return ;
```

  - → Impossible de désallouer.
  - ⇒ Fuites mémoire et dégradation de performances avec le temps.

- **Passage d'arguments par adresse**

« Retourner » plusieurs valeurs

- Parfois une fonction « devrait » fournir plusieurs résultats en sortie.
- Ou un résultat « *composite* ».
- Ex : filtrage d'un point par convolution.
  - Retourne 3 composantes : rouge, vert, bleu.
- En C, `return` ne permet de retourner qu'une valeur.
- Utiliser des variables globales n'est pas satisfaisant (récursivité).
- Déclarer une `struct` par fonction est pénible et pas maintenable :
  - Une `struct` pour retourner 2 entiers,
  - Une `struct` pour retourner 3 entiers,
  - Une `struct` pour retourner 2 entiers et 1 flottant,

  ...

Passage d'arguments par adresse

- Pour « *retourner* » plusieurs résultats, une fonction n'a qu'à prendre l'adresse de variables où stocker ses résultats.
- Mode de passage de paramètres appelé « *out* » (cf. `scanf`).

```
int* random_array (int *size) {
  *size = 1 + random () ;
  return (malloc (*size * sizeof (int))) ;
}
```

- La fonction peut aussi utiliser les valeurs initiales de ces arguments avant de les écraser.
- Mode de passage de paramètres appelé « *in/out* ».

```
void scale_point (int *x, int *y, int scale) {
  *x = *x * scale ;
  *y = *y * scale ;
}
```

- Rem : un tableau est désigné par l'adresse de sa 1$^{\text{ère}}$ case, il est forcément passé par adresse (on y reviendra).

Au boulot...

Écrivez une fonction qui tire deux nombres au hasard et les « retourne » par adresse à votre `main` qui les affichera.

Afin de bien voir que ces fonctions communiquent par des adresses, rajoutez dans votre fonction et dans votre `main` l'affichage de l'adresse des variables.

La fonction permettant de générer un entier aléatoire est `rand` dont la documentation peut être obtenue via sa page de `man` en invoquant la commande shell :

```
man 3 rand
```

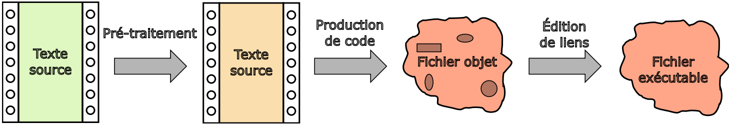
Que remarquez-vous, sur plusieurs exécutions, à propos des valeurs des entiers tirés au hasard ?

Comment pourrait-on y remédier ?

Les « programmes » et leur étapes

Du source au binaire

- « Compilation » : séparée en plusieurs étapes.
- « gcc » = plusieurs programmes (∼ `cpp`, `cc1`, `gas`, `ld`).
- Point de départ : texte.
- Point d'arrivée : binaire exécutable.
- Variations sur la notion « d'exécutable » et de « compilation ».
- On reste sur un modèle à la `C`.
- Schéma global :



- **Pré-processeur**

Pré-traitement / pré-processeur

- Travail de cpp.
- Gère les `#include` et les macros (`#define` et consorts).
- Manipulations purement textuelles.
- Aucune reconnaissance de la structure ni du sens d'un programme `C`.
- Peut s'appliquer sur n'importe quel fichier texte (avec des `#include`, `#define` et consorts).
- Retourne un nouveau texte (sur la sortie standard).

Au boulot...

Reprenez le code source du tout premier programme `joe.c` et appliquez-lui le pré-processeur. Au choix, vous pouvez appeler indifféremment les commandes `cpp` ou `gcc -E`.

Examinez et commentez le résultat.

À quoi peuvent bien servir les lignes de la forme

```
# 885 "/usr/include/stdio.h" 3 4
```

?

Directives de pré-traitement

- `#include`
  - Remplacé textuellement par le contenu du fichier spécifié.
  - `<fichier>` : recherche `fichier` parmi des répertoires par défaut.
  - Possibilité de rajouter un répertoire de recherche avec l'option `-I répertoire` de `gcc`.
  - `"chemin/fichier"` : recherche exactement `chemin/fichier` depuis le répertoire courant.
- `#define`
  - Remplacé textuellement par la chaîne suivant le nom de la macro.
    - `#define ZERO 00`
    - `ZERO␣plus␣ZERO␣=␣␣␣bla␣;` → `00␣plus␣00␣=␣␣␣bla␣;`
  - Possibilité de macros avec des paramètres.
    - `#define SETVAR(v,i) v␣=␣i␣;`
    - `SETVAR(mypi,␣␣3.14)␣;` → `mypi␣=␣␣␣3.14␣;␣;`
    - Pas d'espace avant les parenthèses.
    - Paramètres remplacés par les chaînes effectives lors de l'utilisation.
    - Comportement des espaces « superflus » peut varier selon les versions et les options.

Au boulot. . .

Dans un fichier `foo` définissez une macro `MYMULT` prenant 2 arguments et qui s'expanse en la multiplication de ses 2 arguments.

Dans un fichier `bar`, incluez le fichier `foo` et utilisez votre macro `MYMUL` avec différents arguments (par exemple `0` et `"yop"`, `10` et `3 + 4`, etc.).

Invoquez ensuite le pré-processeur sur le fichier `bar`.

Concluez.

Macros versus `const`

- Macro `#define ZERO 00`
  - Expansion textuelle ⇒ symbole disparaît du code source.
  - Remplace les occurrences de `ZERO` par le texte `00`.
- Définition `const int ZERO = 0 ;`
  - Définit une vraie variable et permet des vérifications de typage.
  - Peut être manipulée comme toute variable (ex. prendre son adresse).
  - Variable possiblement éliminée par les optimisations.

Macros versus fonctions

- Macro `#define MAX(a,b)((a)> (b)? (a) : (b))`
  - Expansion textuelle ⇒ `a` et `b` remplacés par le code source des arguments effectifs.
  - Attention aux effets de bord : `max (++i, j)`
    - Quelle valeur de `MAX (i, j)` pour `i = 1` et `j = 1` ?
  - Duplication de code.
- Fonction `int max (int a, int b){ ... }`
  - Arguments évalués une seule fois.
  - Pas de duplication de code.
  - Permet des vérifications de typage.
  - A une existence dans l'exécutable (pratique pour débug).
  - Mais coûte a priori un appel de fonction.
    - Compilateurs modernes peuvent (se) permettre de l'*inlining*.
- Morale : macros pas diaboliques, leur mauvaise utilisation oui.

- **Production de code**

Production de code

- Génère des instructions propres au microprocesseur cible.
- Change profondément la structure du programme.
- Exploite les informations de type pour sélectionner les instructions.
- En interne : séparée en plusieurs passes intermédiaires.
- Génère du code source assembleur puis appelle le programme d'assemblage.
- Résultat : un fichier binaire non encore exécutable.
- Présence de « trous » : fonctions et variables d'autres fichiers sources, de bibliothèques, etc.

Au boulot. . .

Prenez n'importe lequel des programmes que vous avez déjà écrits (au hasard, celui sur le max des arguments) et appliquez `gcc -S` dessus.

Un fichier assembleur de suffixe `.s` est créé.

Consultez-le et commentez.

- **Édition de liens**

Édition de liens

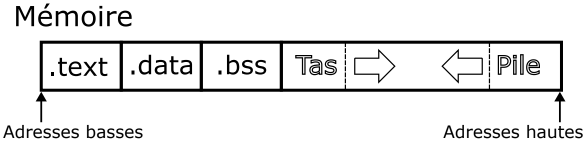
- Regrouper les fichiers objets issus de différents sources.
- Regrouper les fichiers objets issus des bibliothèques statiques.
- « Boucher les trous » en résolvant les références en suspens.
- Fichier exécutable : constitué de différentes sections :
  - code exécutable (`.text`),
  - données en lecture seule (`.rodata`)
  - données globales (ou statiques) non-initialisées ou initialisées à 0 (`.bss`),
  - données globales (ou statiques) initialisées (pas à 0) (`.data`),
  - et bien d'autres encore ...

Au boulot. . .

Compiler n'importe lequel de vos codes sources en un exécutable et invoquez `objdump -h` dessus afin d'obtenir la liste des sections.

Et l'exécution effective ?

- Le reste de l'histoire relève du système d'exploitation.
- Charger le binaire en mémoire.
- Attribuer de la mémoire aux sections pertinentes.



- « Faire pointer » le processeur sur le code.
- Attribuer périodiquement du temps processeur à ce code.
- Morale : compilateur et OS sont intimement liés.

Avertissements, erreurs et plantages

Coup d'œil en arrière

- Dans l'épisode précédent :
  - fonctionnalités (de base) du pré-processeur,
  - étapes de construction d'un exécutable,
  - construction à partir de plusieurs fichiers objets.
- Jouons autour de la compilation séparée (plusieurs fichiers sources).

Au boulot...

Dans un fichier `mkarray.c` définissez une fonction `f` prenant un entier `n` en argument et qui alloue et retourne un tableau de `n` flottants `double`.

Dans un fichier `tstarray.c`, appelez votre fonction `f` pour créer un tableau de taille quelconque (2 par exemple) puis parcourrez le tableau obtenu en écrivant ce que vous voulez comme valeur dans chaque case (vous pourrez aussi y lire pour voir ce que vous y avez écrit).

Attention : Soyez mauvais élève et faites en sorte que le prototype de la fonction `f` soit inconnu dans `tstarray.c`.

Compilez uniquement `tstarray.c` et observez le(s) message(s). N'utilisez pas les options de compilation recommandées précédemment pour une fois (`-Werror` etc.).

Au boulot...

Faites en sorte de corriger ce qui généré l'erreur et l'abandon de la compilation (et juste ça).

Reste-il des messages ?

Y a-t-il un exécutable à faire tourner ?

Si oui, exécutez-le.

Au boulot...

Faites maintenant en sorte que dans `tstarray.c` le prototype de `f` soit connu.

Compilez, exécutez, discutons.

Interaction avec le reste de l'univers

Au boulot…

Écrivez un programme `md.c` prenant en argument un nom et créant un répertoire de ce nom dans le répertoire courant et affiche le nom du répertoire créé.

La fonction de la bibliothèque Unix permettant de faire ce travail se nomme (curieusement) `mkdir` et sa documentation est accessible via sa page de `man` :

```
man 2 mkdir
```

Compilez-le en `md.x`.

Code de statut retourné par le `main`

- 0 ≡ « terminé sans erreur ».
- Sinon, signale une erreur rencontrée par le programme.
- Souvent, convention non respectée par les programmeurs
  - (débutant ou plus expérimentés).
- Possibilité d'enchaîner les commandes dans un script *shell* (cf. MO101).
- En `bash`, `$?` = code de retour de la dernière commande exécutée.
- En `bash`, construction conditionnelle `if`.

Au boulot…

Récupérez le script `bash` suivant depuis le Moodle dans « `Archives énoncés/02 zipball-enonce-aroundpgms.zip` ».
(`https://moodle.ip-paris.fr/course/view.php?id=7789#`)

```
#!/bin/bash

DIR=`./md.x`
if [[ "$?" == 0 ]] ; then echo "On va bosser avec $DIR"; else echo "Erreur."; fi

DIR=`./md.x ici/et/la`
if [[ "$?" == 0 ]] ; then echo "On va bosser avec $DIR"; else echo "Erreur."; fi
```

Rendez `script.sh` exécutable (souvenirs de MO101 ?).

Exécutez le script et discutons des comportements observés.

Au boulot. . .

Si besoin, corrigez votre programme.

Les scalaires

- **Les entiers**

Les entiers

- Plusieurs tailles.
- Signés ou non.
- Par défaut : un entier est signé.

	Taille (bits)	Valeur Min	Valeur Max
short	16	-32768	32767
unsigned short	16	0	65535
int	32	-2^{31}	$2^{31}-1$
unsigned int	32	0	$2^{32}-1$
long int	64	-2^{63}	$2^{63}-1$
unsigned long int	64	0	$2^{64}-1$
char	8	N.S.	N.S.
signed char	8	-2^{7}	$2^{7}-1$
unsigned char	8	0	2^{8}

Codage des entiers

- Non signés : codés en base 2.
  - $421 \mapsto 0000000110100101$ (sur 16 bits)
  - $= 2^8 + 2^7 + 2^5 + 2^2 + 2^0$
- Signés : complément à 2.
  - Bit de poids le plus fort représente le signe $(0 \equiv +, 1 \equiv -)$.
  - Opposé : inversion bits et $+ 1$.
  - $4 \longrightarrow -4 : 00100 \rightarrow 11011 + 1 \rightarrow 11100$.

Débordement d'entiers

- Arithmétique entière et modulo la taille des mots machine.
- Pour les exemples suivants, taille = 4 bits.
- ⇒ Perte de plein de « *bonnes* » propriétés mathématiques.
  - Débordement de non signé : $9 + 7 = 16$?
    $1001 + 0111 = 1|0000 \quad =$ sur 4 bits . . .$0000 = 0$
  - Débordement de signé : -8 - 1 = -9 ?
    $1000 - 0001 = 1|1111 \quad =$ sur 4 bits . . .$1111 =$ -1
  - Conversion de signé → non signé : -4 → ?
    $1100 \quad = 12$
  - Conversion de non signé → signé : 15 → ?
    $1111 \quad =$ -1
  - Associativité de $*$ et / : (1 / 4) * 4 = (1 * 4) / 4 ?
    1 / 4 = 0.25 En entiers . . . 0. Donc, * 4 → 0
    1 * 4 = 4 En entiers . . . 4. Donc, / 4, → 1

Au boulot…

- Soit le programme suivant :

```c
#include <stdio.h>
int main () {
  int s = -1 ;
  unsigned u = 1 ;

  if (s < u) printf ("-1 < 1\n") ;
  else printf ("1 < -1\n") ;
  return 0 ;
}
```

- Que va-t-il afficher ?
- Pourquoi à votre avis ?

Au boulot...

Contexte : `int` = 4 octets, `short int` = 2 octets.

À quelles condition ces conversions sont-elles sûres ?

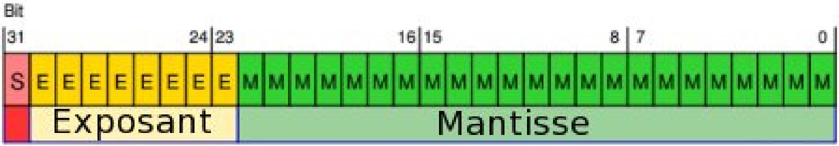
- `int` → `signed int`
- `unsigned int` → `int`
- `int` → `short int`
- `short int` → `unsigned short int`
- `short int` → `int`

- **Les flottants**

Les réels

- Couramment appelés « flottants » (« *floating point numbers* »).
- Toujours signés.
- 2 tailles ⇒ 2 précisions.

	Taille (bits)	Valeur Min	Valeur Max	Précision max
float	24 + 8	$1.1 \cdot 10^{-38}$	$3.4 \cdot 10^{38}$	10^{-6}
double	53 + 11	$2.2 \cdot 10^{-308}$	$1.8 \cdot 10^{308}$	10^{-15}



La réalité des réels : des problèmes bien réels

- Nombres flottants pratiques pour approximer les réels mathématiques.
- Pourtant bien différents de la « beauté » mathématique :
  - Notion de précision :
    - $\Rightarrow$ Tous les réels ne sont pas représentables entre 2 réels.
    - Il existe des « trous » entre 2 réels.
  - Comme pour les entiers : notion de réels min et max représentables.
  - Mauvais conditionnement :
    - Opérations entre grands et petits flottants parfois incohérentes.
    - Ex : 100.0 + 4e+15 = ... 4e+15
  - Arrondis lors des calculs :
    - Test d'égalité == proscrit.
    - Test « modulo un ϵ ».
    - Attention, ϵ n'est pas absolu et dépend du problème !
  - Conversion réel $\rightarrow$ entier : 0, débordement, troncature.

Des réels différents égaux...

thiskillsme.c

```
#include <stdio.h>

int main () {
  float fl1 = 48431.1231 ;
  float fl2 = 48431.1239 ;
  float fl3 = 48431.1250 ;

  printf ("fl1: %f  fl2: %f  fl3: %f\n", fl1, fl2, fl3) ;
  printf ("fl1 == fl2 ? %d\n", (fl1 == fl2)) ;
  printf ("fl2 == fl3 ? %d\n", (fl2 == fl3)) ;
  printf ("fl1 == fl3 ? %d\n", (fl1 == fl3)) ;
  fl1 = fl1 + 0.0001 ;
  printf ("fl1 + 0.0001: %f\n", fl1) ;
  return 0 ;
}
```

```
$ ./thiskillsme.x
fl1: 48431.125000  fl2: 48431.125000  fl3: 48431.125000
fl1 == fl2 ? 1
fl2 == fl3 ? 1
fl1 == fl3 ? 1
fl1 + 0.0001: 48431.125000
```

. . . et des réels égaux différents

thiskillsmeagain.c

```c
#include <stdio.h>
#include <math.h>
#define EPSILON (1e-6)  // Tout petit epsilon...

int main () {
  double fl1 = 0.1 ;
  double fl2 = 0.2 ;
  double fl3 = 0.3 ;
  double fl4 = fl1 + fl2 ;

  printf ("fl1: %f  fl2: %f  fl3: %f  fl4: %f\n", fl1, fl2, fl3,
      fl4) ;
  printf ("fl3 = fl4 ? %d\n", (fl3 == fl4)) ;
  printf ("fl3 ~ fl4 ? %d\n", (fabs (fl3 - fl4) < EPSILON)) ;
  return 0 ;
}
```

```
$ ./thiskillsmeagain
fl1: 0.100000  fl2: 0.200000  fl3: 0.300000  fl4: 0.300000
fl3 = fl4 ? 0
fl3 ~ fl4 ? 1
```

Au boulot. . .

Essayez en OCaml (il est installé sur `info1.ensta.fr`).

Comparez les comportements.

Mémoire, pointeurs et autres joyeusetés

- **Chaînes de caractères**

Au boulot. . .

Soit les deux définitions globales de chaînes :

- `char *s = "IN102";`
- `char t[] = "IN102";`

Écrivez un programme qui affiche les valeurs de `s` et de `t`, compilez-le, exécutez-le.

Quel est le résultat, vous surprend-il ?

Au boulot. . .

Modifiez maintenant votre programme en déclarant 2 fonctions :

- `void print_s (char *str);`
- `void print_t (char str[]);`

qui affichent la chaîne qu'elles reçoivent en argument et utilisez `print_s` pour afficher `s` et `print_t` pour afficher `t`.

Compilez-le, exécutez-le.

Quel est le résultat, vous surprend-il ?

Au boulot. . .

Modifiez maintenant votre programme en appelant `print_s` pour afficher `t` et `print_t` pour afficher `s`.

Compilez-le, exécutez-le.

Quel est le résultat, vous surprend-il ?

Et alors quoi ?

- On vous a toujours dit :
  - « Tableaux et pointeurs c'est la même chose ».
  - « Un tableau c'est l'adresse de sa première case donc c'est un pointeur qui vaut l'adresse de sa première case ».
- Si vous avez eu de la chance : « ... presque la même chose ».
- Dans la plupart des cas d'utilisation c'est vrai.
- Sauf que... la « plupart » ce n'est pas « tous » !
- $\exists \neq \forall$;-)

Au boulot...

Dans un fichier `tab.c` définissez le tableau (la chaîne)

```
char t[] = "IN102";
```

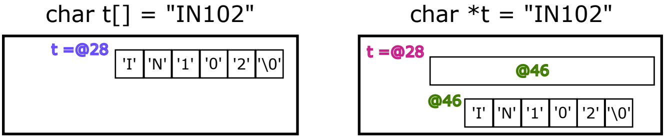
Dans un fichier `maintab.c` déclarez `t` comme étant

```
extern char *t
```

Puis écrivez un `main` qui affiche la chaîne contenue dans `t`.

Compilez, exécutez, appréciez l'équivalence.

Différence d'organisation mémoire



- `char t[] = "IN102"`
  - Crée un tableau rempli avec les caractères.
  - Emplacement de `t` connu à la compilation (ex `@28`).
  - `t[2]` ≡ `*(28 + 2)`.
- `char *t = "IN102"`
  - Créé un pointeur.
  - Emplacement de `t` connu à la compilation (ex `@28`).
  - Et crée un tableau anonyme rempli avec les caractères.
  - `t` contient l'adresse de ce tableau anonyme (ex `@46`).
  - `t[2]` ≡ `*(*28 + 2)`.

Mais alors ?

- Pourquoi donc nos premiers tests de mélanges ont fonctionné ?
- Quand on appelle une fonction :
  - évaluation des arguments,
  - transfert à l'appelé par copie (dans registres ou pile).
- À chaque fois on a copié une adresse (en registre ou pile).
- ⇒ Les fonctions recevaient bien toujours un pointeur contenant l'adresse du tableau.
- Le compilateur connaissait vraiment l'organisation mémoire.
- Il a choisi sciemment d'appeler en « copiant » la valeur de `t` ou `*t`
  - `char t[]` : il a copié l'adresse où il a mis le tableau `t` (aucune autre interprétation raisonnable possible)
  - `char *t` : il a copié la valeur de `t` (comportement habituel)

Autre différence « superficielle »

```c
#include <stdlib.h>   /* Pour NULL. */

int main () {
  char *s = "IN102" ;
  char t[] = "IN102" ;

  s = NULL ;
  t = NULL ;
  return 0 ;
}
```

```
$ gcc badarray.c
badarray.c: In function 'main':
badarray.c:8:5: error: assignment to expression with array type
    8 |   t = NULL ;
```

- `s = NULL` accepté.
- `t = NULL` refusé.
- Quel sens cela aurait d'affecter une adresse constante ?

Au boulot...

Reprenez votre premier programme qui affichait simplement les chaînes `s` et `t` dans son `main`.

Laissez les deux définitions de chaînes en variables globales (plus simple pour la suite mais ne changera rien au comportement).

Remplacez l'une des deux par `"IN1xy"` pour pouvoir les différencier plus tard. Par exemple :

```
char t[] = "IN1xy" ;
char *s = "IN102" ;
```

Rajouter une affectation changeant un caractère dans `t` puis imprimer-la. Faites de même pour `s`.

Compilez, exécutez, appréciez.

Au boulot...

Pour comprendre, regardons un peu l'assembleur généré...

Compilez votre source avec `gcc -S`. Un fichier assembleur de suffixe `.s` est créé.

Ouvrez-le et regardons.

Explication

```
  .file  "ptrvsarray2.c"
  .text
  .globl  s
  .section  .rodata
.LC0:
  .string  "IN102"
  .section  .data.rel.local,"aw"
  .align 8
  .type  s, @object
  .size  s, 8
s:
  .quad  .LC0
  .globl  t
  .data
  .type  t, @object
  .size  t, 6
t:
  .string  "IN1xy"
```

- Souvenirs d'histoires de sections `.text`, `.data`, `.rodata`, `.bss` ?
- Chaîne `"IN102"` (`char *s`) en section `.rodata` :
  - Tableau anonyme lié au pointeur mis en zone lecture seule.
  - ⇒ Impossible de modifier son contenu.
- Chaîne `"IN1xy"` (`char t[]`) en section `.data` :
  - Tableau explicite mis en zone lecture / écriture.
  - ⇒ Contenu modifiable.

Possibles optimisations

```
/* shared.c */
char tab2[] = "IN102" ;     char *str2 = "IN102" ;
```

```
/* sharedmain.c */
#include <stdio.h>
char tab1[] = "IN102" ;     char *str1 = "IN102" ;
extern char tab2[] ;        extern char *str2 ;

int main () {
  printf ("tab1: %p, tab2: %p\n", tab1, tab2) ;
  printf ("str1: %p, str2: %p\n", str1, str2) ;
  return 0 ;
}
```

- gcc 11.3.0 (Ubuntu) :

  ```
  tab1: 0x5566fc2e3020, tab2: 0x5566fc2e3010
  str1: 0x5566fc2e100a, str2: 0x5566fc2e1004
  ```

- clang 14.0.3 (macOS) :

  ```
  tab1: 0x1026fc010, tab2: 0x1026fc000
  str1: 0x1026f7f88, str2: 0x1026f7f88
  ```

Explication

- Selon le compilateur, chaînes statiques peuvent être partagées.
- Même contenu ⇒ même adresse.
- But : optimisation empreinte mémoire.
- Aurait été pareil si `str1` avait été variable locale au `main`.
- Sous Linux avec `gcc -O1` ⇒ optimisation apparaît.
- Dépend donc aussi des options de compilation.
- Est-ce correct et pourquoi ?

Conclusion sur les chaînes

- Chaînes et tableaux presque toujours la même chose.
- Quand plantage en manipulant des chaînes, vérifier :
  - Mémoire bien allouée (pas de pointeur nul) ?
  - Sortie de la zone allouée (débordement de chaîne) ?
  - Chaîne est mutable ou immutable ?
  - Mélange `char*` et `char[]` ?

- **Retour sur les tableaux**

Au boulot. . .

Soit le programme suivant :

```
#include <stdio.h>
#define SIZE (4)

int* compute_array () {
  int t[SIZE] ;
  for (int i = 0; i < SIZE; i++) t[i] = i * 10 ;
  return t ;
}

int main () {
  int *t = compute_array () ;
  for (int i = 0; i < SIZE; i++) printf ("%d\n", t[i]) ;
  return 0 ;
}
```

Commentez, expliquez le comportement.

Au boulot...

Corrigez le programme précédent afin qu'il devienne correct.

Au boulot…

Dans un simple `main`, définissez un tableau statique `t1` de 5 `int`.

Définissez un tableau `t2` de 5 `int` alloué dynamiquement.

Définissez un tableau `t3` de `n` `int` en utilisant la variable `int n = 5 ;`

Utilisez `sizeof` pour afficher la taille de ces tableaux.

Tableaux à longueur variable et typedef

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
  int n = 5 ;
  typedef int fiveints_t[n] ;
  fiveints_t t1 ;        printf ("%lu\n", sizeof (t1)) ;
  n = 10 ;
  fiveints_t t2 ;        printf ("%lu\n", sizeof (t2)) ;

  if (atoi (argv[1]) % 2) n = 20 ; /* Et la vérif des arguments ? */
  else n = 30 ;
  typedef int manyints_t[n] ;
  manyints_t t3 ;        printf ("%lu\n", sizeof (t3)) ;
  return 0 ;
}
```

```
$ gcc vla_typedef.c
./a.out 0
20
20
120
$ ./a.out 1
20
20
80
```

- `typedef` finalisé au moment où le contrôle passe dessus.
- Pas uniquement une déclaration : génère du code exécutable.

Au boulot. . .

`sizeof`, `sizeof`, `sizeof` . . .

Est-ce une fonction ?

D'où vient-il ?

Quels sont ses intérêts ?

- **Tableaux statiques à plusieurs dimensions**

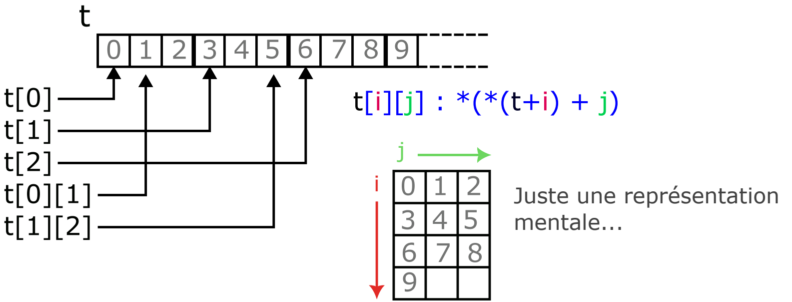
Au boulot...

Soit la définition de tableau à 2 dimensions suivante. Comment peut-on déteminer comment sont rangés en mémoire les différents entiers ?

```
#define NB_LINES (5)
#define NB_COLS (3)

int t[NB_LINES][NB_COLS] = {
  { 0, 1 ,2 },
  { 3, 4, 5 },
  { 6, 7, 8 },
  { 9, 10, 11 },
  { 12, 13, 14 }
};
```

Agencement effectif



- `t[5][3]`,
- `t` est un tableau de taille 5,
- contenant des élements étant des tableaux de taille 3 `int`.

Et on généralise. . .

int t[4][3][2]


```
#include <stdio.h>

int main () {
  int t[4][3][2] = {
    { { 0, 1 }, { 2, 3 }, { 4, 5} },
    { { 6, 7 }, { 8, 9 }, { 10, 11 } },
    { { 12, 13 }, { 14, 15 }, { 16, 17 } },
    { { 18, 19 }, { 20, 21 }, { 22, 23 } }
  };

  for (int i = 0; i < 4 * 3 * 2; i++)
    printf ("%d ", ((int*) t)[i]) ;
  printf ("\n") ;
  return 0 ;
}
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
```

Accéder aux « sous-tableaux »

```
int t[4][3][2] = /* tt */ {
           { { 0, 1 },            { 2,          3 },  { 4, 5   } },
  /* ttt*/ { { 6, 7 }, /*tttt */ { 8, /* v */ 9 },  { 10, 11 } },
           { { 12, 13 },          { 14,         15 }, { 16, 17 } },
           { { 18, 19 },          { 20,         21 }, { 22, 23 } }
};
```

- Déclarer des pointeurs
  - Parenthèses importantes (cf. diapo suivante).
- `int (*tt)[3][2] = t ;`
  - `tt` est un pointeur vers une zone contenant des `int[3][2]`.
- `int (*ttt)[2] = t[1] ;`
  - `ttt` est un pointeur vers une zone contenant des `int[2]` (des « paquets » de 2 `int`).
- `int *tttt = ttt[1] ;`
  - `tttt` est un pointeur vers des `int` (un « paquet » de 2 `int`).
- `int v = tttt[1] ;`
  - `v` est un entier... qui vaut 9.

Importance des parenthèses

```
#include <stdio.h>

int main () {
  int a = 10, b = 20 ;
  int *tab_ptr [2] = { &a, &b} ; /* Tableau de 2 pointeurs sur int. */
  int t[2] = { a, b } ;
  int (*ptr_tab)[2] = &t ;       /* Pointeur vers tableau de 2 int. */

  printf ("&a: %p, &b: %p\n", &a, &b) ;
  printf ("%p, %p\n", tab_ptr[0], tab_ptr[1]) ;
  printf ("%d, %d\n", (*ptr_tab)[0], (*ptr_tab)[1]) ;
  return 0 ;
}
```

- `int (*ptr_tab)[2] = &t ;`
  - `ptr_tab` est un pointeur vers un tableau.
  - Encore un cas ou pointeur et tableau c'est presque la même chose.
- `printf (..., (*ptr_tab)[0], ...) ;`

```
&a: 0x16bb1b428, &b: 0x16bb1b424
0x16bb1b428, 0x16bb1b424
10, 20
```

Jardinage mémoire et corruption de `malloc`

Au boulot...

Attention : À faire impérativement tourner sur la machine Linux `info1`.

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE (9)

int main () {
  int *t = malloc (SIZE * sizeof (int)) ;
  if (t == NULL) return 1 ;
  for (int i = 0; i <= SIZE; i++) t[i] = i * 10 ;
  for (int i = 0; i <= SIZE; i++) printf ("%d\n", t[i]) ;
  free (t) ;
  return 0 ;
}
```

- Que fait ce programme ?
- Quel va être son affichage ?

Au boulot...

- Modifiez le programme pour que la taille de mémoire allouée soit de 10 `int`.
- Que se passe-t-il ?

Taille effectivement allouée

- `malloc (n)` alloue au moins *n* octets.
- Contraintes internes ⇒ allocation souvent légèrement supérieure.
- ⇒ Mémoire effectivement disponible un peu plus grande.
- Si écriture au-delà de la zone effectivement disponible, destruction des informations administratives nécessaire à la gestion mémoire.
- ⇒ Bugs mémoire latents possibles et difficiles à trouver.

Interlude : opérations bit-à-bit

Au boulot. . .

Complétez les tables de vérité suivantes :

et	0	1
0		
1		

ou	0	1
0		
1		

ou-excl	0	1
0		
1		

	non
0	
1	

Opérateurs bit-à-bit de C

- Opérateur unaire : applique l'opérateur logique sur chacun des bits de l'opérande séparément.
- Opérateur binaire : applique l'opérande logique sur chaque couple de bits de même rang séparément.
- Uniquement sur des scalaires.

non	et	ou	ou-excl
~	&	|	^

- Utile lors de manipulation (très) bas-niveau :
  - Interaction avec le matériel (dernière séance).
  - Encodage compact d'information.

Décalages

- `x << n` : décale les bits de x vers la gauche n fois.
- `x >> n` : décale les bits de x vers la droite n fois.
- n doit être $\geqslant 0$ (sinon comportement indéfini).
- `<<` : insère des 0's à droite.
- `>>` :
  - Si x non signé ou positif : insère des 0's à gauche.
  - Sinon, dépendant du compilateur.
    - Généralement, insertion du même bit que celui de poids fort.
  - Décalage dit « arithmétique » et non « logique ».
- `x << n` : équivalent à $x \times 2^n$.
- `x >> n` : équivalent à $\lfloor x/2^n \rfloor$.

- Opérations très rapides.
- Utilisées par les compilateurs pour remplacer / et $\times$ avec des constantes entières.

Au boulot...

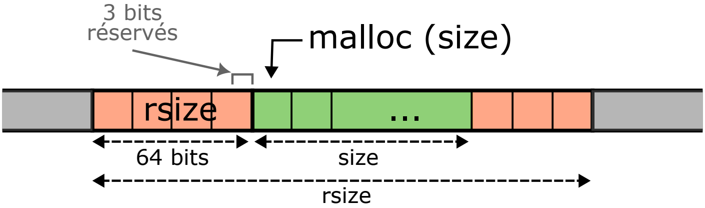
- Vérifier si un entier est impair.
- Arrondir un entier au multiple de 8 inférieur ou égal le plus proche.
- Constante entière avec 8ème bit (partant de la droite) à 1.
- Récupérer le 6ème bit (partant de la droite) d'un entier.
- Vérifier si un entier non signé est une puissance de 2.

Retour sur les dessous de `malloc`

Applications des manipulations bit-à-bit

- On y reviendra plus tard en interagissant avec l'électronique.

- Retour sur les dessous de `malloc`.
- Sous `Linux` + `gcc 11.3.0` + architecture `X64` :
  - Taille du bloc réellement alloué : 64 bits situés 8 octets avant l'adresse retournée par `malloc`.
  - Taille : inclut ces 64 bits.
  - Les 3 bits de poids faible sont à ignorer :
    - ► utilisés pour d'autres informations (compacité).



Au boulot…

Dans le fichier `malloc_hack1.c`, écrivez un programme qui effectue une allocation dynamique avec `malloc` (au hasard, 640 `int`), affiche le nombre d'octets demandés et le nombre d'octets réellement présents dans le bloc alloué.

Indication : Le champ de taille (64 bits) a pour type `size_t`.

Surprises : transtypage et arithmétique de pointeurs

- Besoin de soustraire une constante d'un pointeur.
- `int *t`, $\mathtt{t} \pm n \equiv t \pm n \times \mathtt{sizeof}\ (\mathrm{int})$.
- Addition / soustraction d'une constante : taille objet pointé.
- Pour compter en octets : forcer le pointeur en « pointeur sur `char` ».
- Transtypage : `((char*)t) + ...`
- Attention : dangereux car sans aucune sécurité, vérification.
- Seules opérations légales : pointeur $\pm$ constante entière.
- Pas de $\times$, pas 2 pointeurs, pas de flottants...
- `t[i]` $\equiv$ `*(t + i)`.

Au boulot. . .

Dans le fichier `malloc_hack2.c`, étendez votre programme précédent afin d'écrire sur tous les octets de la zone réellement allouée. Puis faites une nouvelle allocation avec `malloc`.

Attention : Souvenez-vous que la taille trouvée compte les 64 bits où elle est inscrite.

Normalement tout doit bien se passer.

Au boulot...

Maintenant écrivez quelques octets plus (trop) loin et comparez le résultat de l'exécution.

Explication

- `malloc` maintient une liste chaînée des blocs libres.
- Informations « administratives » de chaînage et de taille aussi dans la mémoire.
- Écriture sur cette zone de données « administratives »
  - $\Rightarrow$ Corruption taille du prochain bloc libre.
  - $\Rightarrow$ `malloc` ne peut plus s'y retrouver.
- Sans allocation subséquente, aucun plantage.

Champs de bits (bitfields)

Au boulot. . .

Soit une date au format $j : mm : aa$.

Proposez une structure de données permettant de grouper ces 3 informations ensemble.

Quelle est la taille de cette structure de données ?

Au boulot. . .

Quel est le nombre minimal de bits nécessaires pour représenter une telle date ?

Au boulot. . .

À coup de manipulations bit-à-bit proposez les fonctions qui permettent, en encodant une date sur un `unsigned short`, de :

- retourner le jour d'une date,
- retourner le mois d'une date,
- modifier le mois d'une date.

Note : Pour simplifier vos calculs et conversions binaire → décimal (ou hexadécimal), commencez l'encodage avec le jour aux bits de poids les plus forts.

Champs de bits

- Manipulation de bits par décalages, `&`, `|`, `~` peu lisible.
- Idée : nommer les bits individuellement ou « par paquets ».
- Interpréter chaque paquet avec un type entier spécifié.
- Accès aux « paquets » selon leur nom.
- Permet d'optimiser l'espace en groupant des données.

Utilisation d'un champs de bits

```
struct t {
  type1 nom1 : nb-bits1 ;
  type2 nom2 : nb-bits2 ;
  type3      : nb-bits3 ;
  type4 nom4 : nb-bits4 ;
};
```

- Chaque `type`$_i$ indique comment considérer le « paquet » de bits.
- Chaque `nb-bits`$_i$ indique le nombre de bits du « paquet ».
- « Paquets » de bits « tassés » consécutivement.
- Possibilité de « bourrage » avec des champs anonymes.

- Si `type`$_i$ pas de la même taille : taille totale et agencement réel un peu compliqués à déterminer.
- Idem si nombre total de bits ne tient pas sur 1 `type`$_i$.
- Ordre allocation des bits (forts / faibles) non spécifié.
- On ne rentre pas dans les détails...

Au boulot. . .

Récrivez votre programme précédent en tirant partie des champs de bits.

Créez une variable de ce type et initialisez-la à une date de votre choix.

Affichez-là en hexadécimal.

Quelle est désormais la taille d'une date ?

Encoder des types somme

Types somme simple

- En `OCaml` : `type col_t = Red | Green | Blue | Cyan`
- En C : `enum col_t { Red, Green, Blue, Cyan } ;`
- En `OCaml` :
  `match c with Red -> 0 | (Green | Blue) -> 1 | _ -> 2`
- En C :

```
switch (c) {
  case Red : return 0 ; break ;
  case Green :  /* Fallthrough really wanted here. */
  case Blue : return 1 ; break
  default: return 2 ; break ;
  }
```

- Pas de `break` entre cas `Green` et `Blue`.
- *Fallthrough* : au moins un commentaire pour signifier la volonté !

`case` et `break`

- Structure en apparence régulière : `case break` ... `default break`.
- Réalité :
  - `case` correspondent à des étiquettes.
  - Peuvent être mis n'importe comment dans la portée du `switch`.
  - `break` non obligatoires.
  - Peuvent être mis n'importe comment dans la portée du `switch` ou d'une boucle.
    - On y reviendra.
- Compilateurs malins :
  - table de sauts,
  - tests en cascade,
  - voire des choses beaucoup plus subtiles
    - tests d'intervalles, *hash*, comparaison dichotomique, etc.

Cas pathologique

```
#include <stdio.h>

enum t { A, B, C, D } ;

int f (enum t x) {
  int res = 0 ;
  switch (x) {
  case A : printf ("A\n") ;
    if (x == A) {
      printf ("x == A\n") ;
    default:
      printf ("default\n") ;
    }
    res = 1 ;
    break ;
  case B :
    for (;;) break ;
    printf ("B\n") ;
    res = 2 ;
    break ;
  case C : res = 3 ;
  }
    return res ;
}

int main () {
  printf ("%d\n", f (A)) ;
  printf ("%d\n", f (B)) ;
  printf ("%d\n", f (C)) ;
  printf ("%d\n", f (D)) ;
  return 0 ;
}
```

```
$ gcc -Wall foo.c
$ ./a.out
A
x == A
default
1
B
2
3
default
1
```

Type somme complexe

- En `OCaml` : `type t = B of bool | F of float`
- En `C` : `enum` pas suffisant.

  Solution 1 :
- Définir un `enum` pour signifier `B` ou `F`.
- Définir une `struct` avec 3 champs :
  - `enum`,
  - `bool`,
  - `float`.
- Utiliser le champ approprié `bool` ou `float` en fonction de la valeur du champ `enum`.

Au boulot...

Appliquez la technique ci-dessus pour encoder le type somme en `C`.

Écrivez une fonction qui permet d'afficher la valeur « somme » reçue en argument.

Quelle est la taille d'une valeur de ce type ?

Type somme complexe : meilleure solution

- Les champs `bool` et `float` jamais pertinents en même temps.
- Dommage de gaspiller de la place.
- Plutôt qu'un « et » de champs, on voudrait un « ou ».

```
union nom {
  type_i nom-champ_i ;
};
```

- Taille : la plus grande de tous les $champ_i$.
- Chaque $\texttt{champ}_i$ représente la même zone mémoire.
- Accès à un champ : considère la donnée de la zone avec ce type.
- Initialisation : `{ .nom-champ = ... }`
- Gain de mémoire important si beaucoup de « cas ».

Au boulot. . .

Modifiez votre programme précédent pour utiliser une `union`.

Affichez la taille d'une valeur avec ce nouvel encodage.

Rendre l'union anonyme

- Pénible de devoir nommer l'union puis la struct.
- union uniquement utilisée dans la struct.
- Possibilité de définir l'union anonyme à l'intérieur de la struct.

```
enum tag_t { TBool, TFloat } ;
struct my_sum_t {
  enum tag_t tag ;
  union {
    bool as_bool ;
    float as_float ; } data ;
};
```

Au boulot...

Comment encoderiez-vous la structure des expressions arithmétiques donnée par le type `OCaml` suivant ?

```
type arith_t =
| Number of int
| Ident of string
| Add of (arith_t * arith_t)
| Sub of (arith_t * arith_t)
| Mult of (arith_t * arith_t)
| Div of (arith_t * arith_t)
| Uminus of arith_t
```

On apprécie quand ces types de données sont natifs au langage !

Encoder des exceptions

- **Des exceptions simples**

En OCaml

```
try
  let file1 = open_in_bin "fichier.dat" in
  let res1 = compute_machin ... in
  let res2 = compute_bidule .. in
  ...
with
| Sys_error _ -> ...
| Not_found -> ...
```

- Traitement des erreurs centralisé.

Très souvent en C

```
FILE *in = fopen (...) ;
if (in == NULL) return -1 ;
int *t = malloc (...) ;
if (t == NULL) {
  fclose (in) ;
  return -1 ;
}
int *u = malloc (...) ;
if (u == NULL) {
  free (t) ;
  fclose (in) ;
  return -1 ;
}
...
free (u) ;
free (t) ;
fclose (in) ;
return 42 ;
```

- Traitement des erreurs dispersé et répété.

Solution d'amélioration : le `goto`

- `goto` toujours honni et rarement enseigné.
- Raison : `goto` n'importe comment ⇒ code non maintenable.
- Et c'est vrai !
- Si sauts toujours vers le point de sortie de la fonction : simule des exceptions simples
  - Exceptions levées et rattrapées dans une même fonction.
- Raisonnable uniquement pour terminer en cas d'erreur.
- Ne doit jamais remplacer les boucles ou autres structures de contrôle.

Après réécriture

```
FILE *in = NULL ;             /* Valeur d'erreur par défaut. */
int *t = NULL, *u = NULL ; /* Valeur d'erreur par défaut. */
int ret_val = -1 ;            /* Valeur d'erreur par défaut. */

in = fopen (...) ;
if (in == NULL) goto cleanup ;
t = malloc (...) ;
if (t == NULL) goto cleanup ;
u = malloc (...) ;
if (u == NULL) goto cleanup ;
...
   ret_vval = ... ;
...
cleanup:
if (u) free (u) ;
if (t) free (t) ;
if (in) fclose (in) ;
return ret_val ;
```

- Initialisation des variables à leur valeur d'erreur.
- Schéma de programmation dite défensive.

Re-avertissement

- Se restreindre au cas particulier de la gestion d'erreurs.
- Ne pas en abuser.
- Jamais de saut en arrière.

- `break` et `continue` dans les boucles sont en fait des `goto`.
- Ne pas les utiliser autant que possible.
- Disperse la condition de fin de boucle en plusieurs endroits.
- Difficile de synthétiser l'argument de terminaison (variant / invariant).

Au boulot…

Réécrivez le programme suivant sans les horribles `continue` et `break`.

```
#include <stdio.h>

int main () {
  for (int i = 0; i < 10; i++) {
    if (i == 3) continue ;
    if (i == 7) break ;
    printf ("%d\n", i) ;
  }

  return 0 ;
}
```

- **Vers de vraies exceptions**

Vers de vraies exceptions

- But : propager et récupérer des erreurs au travers des appels de fonctions.
- `#include <setjmp.h>`
- 2 fonctions, 1 structure de données :
  - `int setjmp (jmp_buf env)`
  - `void longjmp (jmp_buf env, int val)`
  - `jmp_buf`
- `jmp_buf` : environnement d'appel
  ► Mémorise l'état des registres.
- `setjmp` + `if` ∼ `try with`.
- `longjmp` ∼ `raise`.

`setjmp` / `longjmp`

- `int setjmp (jmp_buf env)`
- « Appel initial » retourne 0.
- Remplit la structure de donnée `env` :
  - état des registres au point courant,
  - point dans le code du retour de l'appel.

- `void longjmp (jmp_buf env, int val)`
- Copie `env` dans les registres (restaure l'état sauvegardé).
- Reprend l'exécution juste au point de retour de `setjmp`...
- mais avec comme valeur de retour `val`.
- « On revient dans le passé en changeant l'histoire ».

⇒ Tester le retour de `setjmp` permet de savoir si `longjmp` a été appelée ... depuis que `setjmp` a été appelée.

Au boulot...

Faites l'essai.

Déclarez un `jmp_buf` global.

Écrivez une fonction qui appelle `longjmp`.

Dans votre `main`, appelez `setjmp` en affichant des messages différents selon sa valeur retournée.

Encodage

- Constructeur d'exception levée : valeur donnée à l'appel à `longjmp`.
- Cas du `with` : `switch` sur la valeur retournée par `setjmp`.
- `case 0` : code « protégé » par le `try ... with`.
- Besoin d'autant de variables globales `jmp_buf` que de `try with`.
- Exceptions ne colportent pas de valeurs.
  - Au minimum, besoin d'une variable globale.

Au boulot. . .

Écrivez l'équivalent en `C` du programme `OCaml` ci-dessous.

```
exception Below0 ;;
exception Above0 ;;

let f x =
  if x < 0 then raise Below0 ;
  if x > 0 then raise Above0 ;
  x + 10
;;

let g y =
  try print_int (f y) with
  | Below0 -> print_string "Below 0"
  | Above0 -> print_string "Above 0"
;;

g (-1) ;;
g 0 ;;
g 1 ;;
```

Subtilités

- Sauvegarde du contexte : pas les variables locales.
- Si allocations de ressources entre `setjmp` et `longjmp` : pas libérées
  - ▶ `malloc`, `fopen`, etc. . .
- Variables locales à une fonction faisant `setjmp` :
  - Peuvent avoir (légitimement) changé quand on revient à `setjmp`.
  - Si variables étaient stockées en registres, alors restaurées.

  $\Rightarrow$ Perte de la valeur modifiée.
  - Besoin de les qualifier `volatile`.
  - Signifie : « variable peut changer à tout moment ».

  $\Rightarrow$ Force le compilateur à relire leur valeur à chaque utilisation.

Quand C plus suffisant

Au boulot. . .

Écrivez une fonction qui prend un `unsigned int` en entrée et affiche sa valeur sous forme binaire.

Rajoutez un `main` qui déclare `unsigned int v` (initialisé à 1 par exemple) et qui effectue une rotation de ses bits vers la gauche 1 fois.

Effectuez cette rotation en boucle (au moins 32 fois) en affichant la valeur de l'entier pour bien voir son bit « se déplacer » (si vous l'avez initialisé à 1).

L'opérateur qui manque

- En C, pas d'opérateur de rotation de bits.
- Pourtant, dans *Intel® 64 and IA-32 Architectures Software Developer's Manual, vol. 2 sec. 4-522* :

INSTRUCTION SET REFERENCE, M-U

RCL/RCR/ROL/ROR—Rotate

Opcode**	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
REX + C0 /0 *ib*	ROL *r/m8*, imm8*	MI	Valid	N.E.	Rotate 8 bits *r/m8* left *imm8* times.
D1 /0	ROL *r/m16*, 1	M1	Valid	Valid	Rotate 16 bits *r/m16* left once.
D3 /0	ROL *r/m16*, CL	MC	Valid	Valid	Rotate 16 bits *r/m16* left CL times.
C1 /0 *ib*	ROL *r/m16, imm8*	MI	Valid	Valid	Rotate 16 bits *r/m16* left *imm8* times.
D1 /0	ROL *r/m32*, 1	M1	Valid	Valid	Rotate 32 bits *r/m32* left once.

- Idée : intégrer des encarts (*inlines*) d'assembleur dans le source C.
- Au compilateur de faire le lien avec le reste du code.
- Impose de lui donner suffisamment d'informations.

Encart d'assembleur : syntaxe

- Contexte : `gcc`, architecture Intel x64.

```
asm asm-qualifiers (AssemblerTemplate
                    : OutputOperands
                    [ : InputOperands
                    [ : Clobbers ] ])
```

- *asm-qualifiers* : pour nous seulement `volatile`
  - Empêche l'optimiseur de supprimer nos ajouts d'instructions.
  - *A priori* pas nécessaire dans notre cas, mais *security first*.
- *AssemblerTemplate* : chaîne représentant l'instruction avec des références aux opérandes de *OutputOperands* et *InputOperands*.
- *OutputOperands* : expressions C utilisées en écriture.
- *InputOperands* : expressions C utilisées en lecture.
  - Si lecture et écriture → *OutputOperands*.
- *Clobbers* : décrit les effets de bord, on ignorera ici.
  - E.g. modification mémoire, modification de flags.

Encart d'assembleur : exemple

```
int x = 42 ;
int y ;
asm volatile ("mov %1, %0\n\t" : "=r" (y) : "r" (x) :) ;
asm volatile ("add $8, %0\n\t" : "+r" (y) : :) ;
printf ("%d\n", y) ;
```

- `y = x + 8`.
- `"r"` : signifie valeur est dans un registre (`"m"` serait en mémoire).
- `"=r"` (sorties uniquement) signifie écrase l'ancienne valeur.
- `"+r"` (sorties uniquement) lit et écrase l'ancienne valeur.
- Si plusieurs opérandes : séparées par des virgules.
- %0, %1 ... référencent les opérandes par leur ordre dans *OutputOperands* + *InputOperands*.

Au boulot...

Copiez-collez les quelques lignes de l'exemple précédent dans un `main` digne de ce nom.

Compilez, testez.

Remplacez le `+r` de la seconde instruction par un `=r`.

Re-compilez, re-lancez. Concluez.

Au boulot. . .

Modifiez votre programme (mais gardez-en une copie pour plus tard) pour remplacer vos opérations bit-à-bit par l'instruction assembleur `rol` vue précédemment, dans sa version 32 bits explicite : `roll`.

Examinez bien l'effet de cette instruction pour déterminer la ou les contraintes portant sur la ou les opérandes de l'instruction.

Au boulot...

Compilez avec `gcc -S` et consultez le fichier assembleur généré pour y retrouver votre instruction.

Remarquez que le compilateur a fait le lien entre votre variable C et le registre qu'il lui a assigné.

Au boulot...

Reprenez votre version initiale du programme (ou du moins la version donnée en solution si la vôtre n'utilisait pas les 2 décalages et le *ou* bit-à-bit).

Compilez-la avec `gcc -S`.

Essayez de retrouver le code généré pour votre instruction de rotation.

Que s'est-il passé ?

Au boulot…

Toujours sur cette même version initiale du programme, regardez ce qui a été généré dans la fonction `print_bits`.

Arrivez-vous à retrouver ce qui a été généré pour `printf ("%d", v & (1 << i)? 1 : 0)}` ?

Combien d'instructions ça prend pour la conditionnelle ?

Une instruction sympathique

- Dans *Intel® 64 and IA-32 Architectures Software Developer's Manual, vol. 2 sec. 3-113* :

BT—Bit Test

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F A3 /r	BT *r/m16, r16*	MR	Valid	Valid	Store selected bit in CF flag.
0F A3 /r	BT *r/m32, r32*	MR	Valid	Valid	Store selected bit in CF flag.
REX.W + 0F A3 /r	BT *r/m64, r64*	MR	Valid	N.E.	Store selected bit in CF flag.

- Permet de tester 1 bit et met le résultat dans le « CF ».
- Opérande gauche : entier à inspecter, opérande droite : rang du bit.
- CF (*Carry Flag*) : bit du registre d'état (SR).
- Question : comment récupérer ce bit CF ?
- On a l'impression de revenir au problème initial. . .

Une autre instruction sympathique

- Dans *Intel® 64 and IA-32 Architectures Software Developer's Manual, vol. 2 sec. 4-596* :

SETcc—Set Byte on Condition

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 97	SETA *r/m8*	M	Valid	Valid	Set byte if above (CF=0 and ZF=0).
REX + 0F 96	SETBE *r/m8**	M	Valid	N.E.	Set byte if below or equal (CF=1 or ZF=1).
0F 92	SETC *r/m8*	M	Valid	Valid	Set byte if carry (CF=1).
REX + 0F 92	SETC *r/m8**	M	Valid	N.E.	Set byte if carry (CF=1).

- `set`*cc* met l'opérande destination à 0 ou 1 en fonction du bit de condition dénoté par *cc*.
- Pour le bit CF, *cc* est « c » ⇒ `setc`.
- **Attention** : opérande destination sur 8 bits (« Set byte if ...»).

Au boulot. . .

Modifiez votre programme (1ère ou 2nde version) pour remplacer la conditionnelle ternaire par l'utilisation des deux instructions que vous venez de voir.

Au boulot. . .

Regardez le code assembleur généré pour cette nouvelle version du programme.

Comparez le nombre d'instructions utilisées pour cette nouvelle « conditionnelle ternaire ».

Conclusion

- Encarts d'assembleur pour éventuellement optimiser.
  Attention, le compilateur est souvent plus efficace que nous.
- Mais surtout pour accéder à des instructions indisponibles en C.
- Permet d'accéder directement aux registres.
  - E.g. routine de changement de contexte dans un OS.
- Plus simple que d'écrire tout en assembleur.
  - Le compilateur gère l'interface entre variables C et registres.
  - Mais il faut ne pas se tromper dans les I/O et contraintes.
- Nécessite de connaître les instructions et les contraintes d'architecture.
- Rend le programme dépendant de l'architecture, voire du compilateur.
  - $\Rightarrow$ Moins de portabilité.

Débugger son programme

Quand ça ne marche pas...

- Votre programme plante sec ou ne donne pas le résultat attendu.
  1. Relire votre code et penser très fort :
     - Efficace pour les bugs simples.
  2. Mettre des `printf` pour tracer ce que fait votre code :
     - Très efficace, même pour les bugs compliqués.
  3. Utiliser un débugger pour inspecter finement l'exécution :
     - Peut être long et très technique.
     - Efficace... pour les bugs désespérés (et désespérants).

Le debugger : gdb

- Nombreuses variantes avec ou sans IHM :
  - `gdb`, `xxgdb`, `cgdb`, `lldb` sous macOS...
- Nécessite de compiler avec l'option `-g` de `gcc`.
- Permet de :
  - démarrer un programme,
  - mettre des points d'arrêt,
  - avancer instruction par instruction,
  - survoler les appels de fonctions,
  - inspecter la mémoire,
  - surveiller des variables,
  - et bien d'autres choses encore...
- S'invoque en ligne de commande (avec le nom de l'exécutable).

L'utilisation la plus simple

- Si votre programme fait *"Segmentation fault (core dumped)"* :
  - le relancer via gdb

```
#include <stdlib.h>
int main () {
  int *t = NULL ;
  t[10] = 100 ;
  return 0 ;
}
```

```
$ gcc -g crash.c
$ ./a.out
Segmentation fault (core dumped)
$ gdb ./a.out
Reading symbols from ./a.out...
(gdb) run
Program received signal SIGSEGV, Segmentation fault.
main () at crash.c:4
4   t[10] = 100 ;
(gdb) quit
```

Au boulot...

Faites un test par vous-même avec le programme proposé sur la diapositive précédente.

Clause de non-responsabilité (*disclaimer*)

- Présentation en mode sans interface utilisateur.
- Plus âpre.
- Mais permet de se servir de `gdb` « de base ».

- Illustration de certaines facilités via l'interface `Visual Studio Code`.
- Parfois quand même besoin d'utiliser les commandes « de base ».

Au boulot…

Récupérez le fichier `gdb.c` et pratiquez en même temps que les diapositives avancent.

Lancement

- `gcc -g gdb.c`
- `gdb ./a.out`

- Programme chargé mais pas en cours d'exécution.
- Si besoin de passer des arguments : `set args 10 foo bar 7`

```
$ gcc -g gdb.c
$ gdb ./a.out
Reading symbols from ./a.out...
(gdb)
```

- En attente de commandes...
- Afficher le code autour du point courant : `list` (ou `l`).

```
(gdb) list
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int get_old_size (int i) {
5   return i + 1 ;
6 }
7
8 int main () {
9   int *p = NULL ;
```

Points d'arrêt et exécution

- Point d'arrêt : instruction où arrêter l'exécution.
  - Instruction au point d'arrêt non encore exécutée.
- Mettre un point d'arrêt
  - `break` n° ligne
  - `break` fichier:n° ligne
  - `break` nom fonction
  - `break` fichier:nom fonction
- Lancer le programme : `run`

```
(gdb) break main
Breakpoint 1 at 0x1188: file gdb.c, line 9.
(gdb) run
Starting program: a.out

Breakpoint 1, main () at gdb.c:9
9   int *p = NULL ;
(gdb)
```

Gestion des points d'arrêt

- Liste des points d'arrêt : `info break`
- Supprimer un point d'arrêt : `delete` n° pt arrêt
- Désactiver un point d'arrêt : `disable` n° pt arrêt
- Ré-activer un point d'arrêt : `enable` n° pt arrêt

```
(gdb) break gdb.c:14
Breakpoint 2 at 0x5555555551b8: file gdb.c, line 14.
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000555555555188 in main at gdb.c:9
breakpoint already hit 1 time
2       breakpoint     keep y   0x00005555555551b8 in main at gdb.c:14
(gdb)
```

Continuer l'exécution

- Instruction suivante :
  - next (« survol » : appel fonction ≡ 1 instruction)
  - step (descend dans appel fonction)
- Continuer → prochain point d'arrêt :
  - cont

```
(gdb) frame
#0  main () at gdb.c:9
9   int *p = NULL ;
(gdb) step
10   int size = get_old_size (-1) ;
(gdb) next
12   if ((size = 0))
(gdb) cont
Continuing.

Breakpoint 2, main () at gdb.c:14
14   for (int i = 0; i < 16; i++)
```

Au boulot. . .

Vous allez redémarrer le programme en cours de débug pour voir la différence entre `step` et `next`.

Pour redémarrer, invoquez la commande `kill`. Les points d'arrêt restent en place et vous pouvez faire `run` à nouveau pour que le programme s'arrête au premier point d'arrêt.

Invoquez `step` 2 fois. Que constatez-vous comme différence par rapport à la session précédente ?

Refaites `cont` pour revenir au même point que là où vous étiez lors de la précédente session (i.e. ligne 14).

Inspecter les variables

- print expression C.

```
(gdb) step
15      p[i] = 2 * i ;
(gdb) print i
$3 = 0
(gdb) print p
$4 = (int *) 0x0
(gdb) print *p
Cannot access memory at address 0x0
(gdb)
```

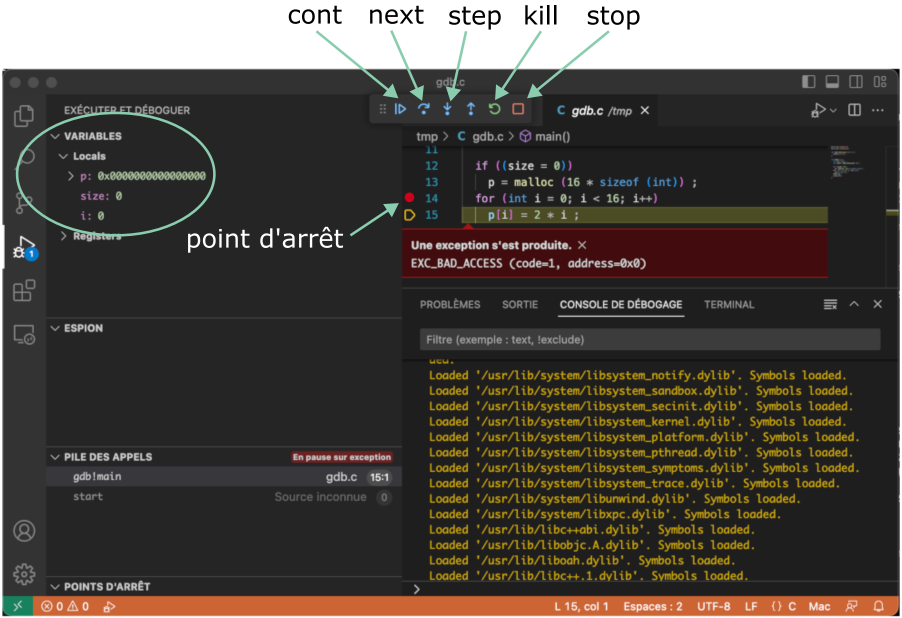
- Oups, p est nul !
- Re-oups, *p se passe plutôt mal !

Quelle est la raison du problème ? Corrigez le code. Pourquoi pas d'avertissement à la compilation ?

Avec une interface plus pratique

- Greffons de `Visual Studio Code` permettent une intégration :
  - *extension C/C++*
  - *C/C++ expansion pack*
- Visualisation du code directement dans l'éditeur.
- Lancement du programme par menu :
  - Exécuter > Démarrer le déboggage.
- Boutons pour `step`, `next` et consorts.
- Poser points d'arrêt par clic droit dans le code.
- Affichage des variables visibles en onglets.
- Possibilité d'invoquer des commandes `gdb` arbitraires :
  - Zone de saisie en bas à droite,
  - Faire précéder la commande par +.

Fenêtre Visual Studio Code



Au boulot. . .

Il vous est donné un code source `mean.c` truffé de bugs.

On attend de ce programme qu'il prenne en argument un nombre de valeurs numériques (**flottants** à saisir), demande à l'utilisateur de saisir ces valeurs une par une, calcule et affiche la moyenne, la valeur minimale et la valeur maximale de valeurs.

En terme d'algorithme, le programme alloue un tableau dans lequel il va stocker sous forme de flottants les valeurs, il fait saisir ces valeurs à l'utilisateur et les stocke dans le tableau, puis en parcourant ce tableau il calcule les `mean`, `min` et `max`. Il libère ensuite ce tableau et affiche ses résultats. Le code est documenté et commente ce processus.

Votre travail dans cet exercice consiste à débugger ce programme avec `gdb` pour en faire une version qui fonctionne conformément à ce que l'on attend de lui.

Indication : Il y a en tout 5 erreurs.

Point d'arrêt conditionnel

- Point d'arrêt avec `break` : identifié par un numéro.
- Lier une condition d'activation au point : `cond` *npt expression*
- Exemple :

```
$ gcc -g mean.c -o mean.x
$ gdb ./mean.x
(gdb) break 44
Breakpoint 1 at 0x13c3: file mean.c, line 44.
(gdb) cond 1 (array_size != 2)
(gdb) set args 1 2 3 4
(gdb) run
Starting program: mean.x 1 2 3 4
Breakpoint 1, main (argc=5, argv=0x7fffffffea38) at mean.c:44
44    if (argc != 2) {
```

Au boulot. . .

Testez les points d'arrêts conditionnels sur le programme.

Watchpoint

- Point d'arrêt : arrêter sur une ligne précise.
- Conditionnel : ligne précise + condition à cette ligne.
- Comment s'arrêter là où une variable est utilisée ?

- « Point d'arrêt » sur une variable : `watch` *variable*.
- Variable doit être visible au moment du `watch`.
- 3 modes de surveillance :
  - `watch` : arrête l'exécution en cas d'écriture dans la variable.
  - `rwatch` : arrête l'exécution en cas de lecture de la variable.
  - `awatch` : arrête l'exécution en cas de lecture ou d'écriture de la variable.

Au boulot. . .

Testez les *watchpoints*.

Juste après avoir chargé le programme dans `gdb`, essayez de mettre un `watch` sur `array_size` et voyez ce qu'il se passe.

Mettez ensuite un point d'arrêt au début du `main`, lancez l'exécution et re-tentez de mettre un `watch` sur `array_size`.

Continuez l'exécution avec `cont` et observez le comportement.

Programmation bas niveau et interaction matérielle

Motivations

- Aspects avancés de `C` abordés jusqu'à présent.
- Intéressant pour comprendre le fonctionnement profond des programmes.
- Intéressant pour comprendre le fonctionnement profond de nos bugs.
- $\Rightarrow$ Peut-être aussi pour éviter nos bugs !

- Mais pas que : ça peut aussi servir autrement.
- Illustration dans le cadre de l'embarqué où `C` est très adapté.
- Nécessite souvent d'interagir avec l'électronique.

- Montrer que la programmation permet de (finit toujours par) contrôler des mécanismes physiques.
- Montrer que les liens entre logiciel et matériel (la physique) n'ont rien de magiques.

- **Présentation et premiers pas**

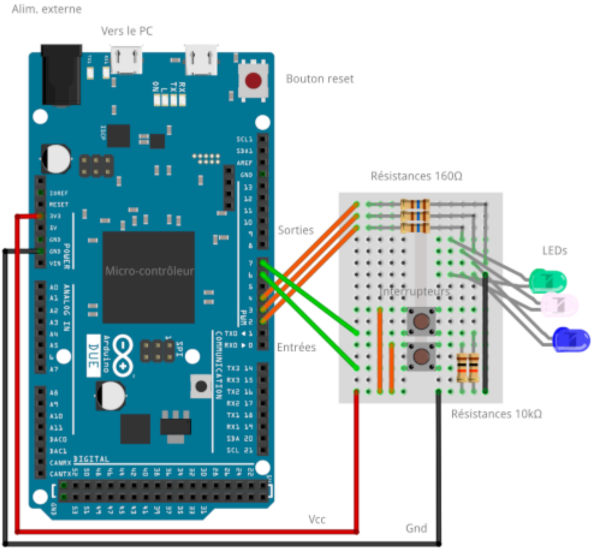
La plate-forme

- Utilisation d'une carte `Arduino`.
- Comprend un micro-contrôleur, de la mémoire, des broches de connexion.
- Micro-contrôleur $\sim$ microprocesseur + contrôleur d'entrées/sorties.
- Permet de contrôler logiciellement les niveaux électriques des broches :
  - en entrée,
  - en sortie,
  - numériquement (allumé / éteint),
  - analogiquement (tension variable).
- Comporte d'autres périphériques internes :
  - minuteurs
  - chiens de garde
  - convertisseurs analogique $\leftrightarrow$ numérique,
  - bus de communication, ...
- Généralement pas de système d'exploitation et programme unique en exécution.

La micro-contrôleur

- `SAM3x8E` (`SAM3X ARM Cortex-M3`).
- Architecture ARM 32 bits $\Rightarrow$ RISC.
- Fréquence : 84 MHz.
- Mémoire de stockage (flash) : 512 Ko.
- Broches E/S numériques : 54.
- Broches analogiques (E) : 12.
- Broches analogiques (S) : 2.
- RAM : 96 Ko.
- « Subtile » différence de puissance avec votre PC.

Le montage électronique



L'environnement de développement

- Disponible sur `https://www.arduino.cc/en/software`.
- Normalement déjà installé en avance sur votre machine.

- Permet d'éditer du code source.
- Permet de compiler avec une version dédiée de `gcc`.
- Permet d'envoyer le binaire sur la carte via un *loader*.

- Possibilité d'utiliser un éditeur de texte externe plus convivial.

Structure générale d'un programme

- Au minimum ces 2 fonctions :
  - `void setup (){ ... }` : exécutée 1 seule fois au démarrage.
  - `void loop (){ ... }` : exécutée en boucle infinie.
- Véritable programme principal (caché) :

```
#include <arduino.h>
autres fonctions () { ... }
void setup () { ... }
void loop () { ... }
int main () {
  setup () ;
  while (1) loop () ;
  return 0 ;  /* N'arrive jamais ici bien évidemment. */
}
```

Au boulot...

Sélectionnez l'entrée de menu Fichier > New Sketch.

Félicitations, vous avez créé sans rien écrire votre premier programme vide.

La toute première fois que vous lancez l'environnement de développement il faut sélectionner le type de carte.

Sélectionnez Outils > Carte > Gestionnaire de carte.

Dans la fenêtre de gauche, installez le module `Arduino SAM Boards (32-bits ARM Cortex-M3)`.

Branchez la carte sur un port USB du PC et sélectionnez, dans le menu déroulant Sélectionner une carte, l'entrée Arduino Due (Program...).

Cliquez sur le bouton en forme de flèche en haut à gauche pour téléverser le programme sur la carte.

Le programme tourne désormais (à ne rien faire) sur la carte.

- **Entrées / sorties**

Affichage pour débugger

- Communication bidirectionnelle via le port USB en mode série.
- Envoi/réception messages à/depuis un PC (terminal).
- Utile pour débugger… quand c'est possible.
- Syntaxe des fonctions parfois un peu bizarre (`C++`).

- `void Serial.begin (speed)` :
  - Initialiser communication avec à une vitesse donnée.
  - Pour nous : 9600 (bits/s).
  - Une seule fois dans `setup`.
- `uint32_t Serial.println (val)` :
  - envoie pour « affichage » la chaîne représentant l'expression `val` avec `'\n'` final.
  - `val` : peut être de type `char`, `int`, `float`, `char*`…
- `uint32_t Serial.print (val)` : comme `println` sans `'\n'` final.

Au boulot...

Ajoutez, à votre programme « vide », l'envoi d'un message de votre choix à destination du PC connecté à la carte.

Compilez et téléversez sur la carte.

Pour accéder aux messages envoyés par la carte sur le port USB, sélectionnez le menu Outils > Moniteur série.

Une sous-fenêtre s'ouvre dans laquelle les messages s'affichent.

Si l'affichage est incohérent, assurez-vous que la vitesse de 9600 est bien sélectionnée dans le popup à droite de la sous-fenêtre.

Au boulot. . .

Dans un contexte ou le débug est impossible car la communication ralentirait trop, ou quand la communication série est impossible (ça existe), comment pourrait-on faire autrement ?

Broches d'entrées/sorties

- Sans E/S, MCU inutile : impossible de discuter avec « l'extérieur ».
- Broches du MCU physiquement connectées aux broches de la carte.
- Mode lecture ou écriture permet des échanges électriques.
- Pour nous, broches numériques (*digital*) :
  - 0 V (« faux », « `false` », « off ») : `LOW`
  - 3.3 V (« vrai », « `true` » « on ») : `HIGH`
- Configuration du mode d'une broche :
  - `void pinMode (uint32 t pin, uint32 t mode)`
  - `pin` : numéro de la broche
  - `mode` : `OUTPUT` ou `INPUT`
  - Généralement 1 fois pour toutes dans `setup`.

Mode sortie

```
#define PIN_LED0 (4)
...
pinMode (PIN_LED0, OUTPUT) ;
...
```

- Écriture sur une broche :
- `void digitalWrite (uint32 t pin, uint32 t value)`
- `value` :
  - `LOW` : niveau électrique à 0 V.
  - `HIGH` : niveau électrique à 3.3 V.

Au boulot. . .

Faites en sorte que la LED bleue clignote toutes les secondes.

La fonction `void delay (unsigned long ms)` permet de suspendre l'exécution pendant `ms` millisecondes.

Variables locales statiques

- Problème peut-être rencontré : LED ne change pas.
- Si variable d'état locale à la fonction, réinitialisée à chaque appel.
- Souvenez-vous : `loop` appelée sans fin par `main`.
- Solution 1 : variable d'état globale
  - Inconvénient : modifiable par toute fonction du fichier.
  - Nuit à la maintenabilité, la modularité et la sûreté.
- Solution 2 : variable locale statique

```
void loop () {
  static bool led_state = HIGH ;
  ...
```

  - Initialisée au premier appel.
  - Conserve sa valeur d'un appel à un autre.
  - Accessible uniquement depuis la fonction où définie.
  - En fait : variable globale mais restriction de porté par le compilateur.
  - Meilleure modularité.

Mode entrée

```
#define PIN_LED0 (6)
...
pinMode (PIN_LED0, INPUT) ;
...
```

- Lecture sur une broche :
- `int digitalRead (uint32 t pin)`
- Valeur retournée :
  - `LOW` : niveau électrique à 0 V.
  - `HIGH` : niveau électrique à 3.3 V.

Au boulot...

Dans un nouveau programme, lisez l'état de la broche connectée à l'interrupteur le plus près des LEDs et affichez cet état par l'envoi de messages sur le port série.

- **Manipulation d'adresses et de bits**

Mémoire *mappée*

- État des broches : bits de certains registres du MCU.
  - Mettre à `HIGH` : mettre un bit à 1 dans un registre.
  - Mettre à `LOW` : mettre un bit à 1 dans un autre registre.
- Registres physiquement accessibles (« *mappés* », « adressables ») dans l'espace mémoire.
- Chaque registre a son adresse en mémoire.
- Écrire dans un registre : écrire à une adresse particulière.
- Positionner un bit dans un registre : positionner un bit dans l'entier à une adresse particulière.
- Taille de l'entier : dépend de la taille du registre.
- $\Rightarrow$ Intérêt des pointeurs et des manipulations de bits.

Démystifier `digitalWrite`

- Manuel constructeur du MCU (1467 pages) pour adresses et bits.
- *PIO Controller Set Output Data Registers* : mise à `HIGH`.
- *PIO Controller Clear Output Data Registers* : mise à `LOW`.
- Broches réparties entre plusieurs registres de 32 bits.
- Bits numérotés à partir de 0.

Broche	Registre	Adresse	Bit
2	`PIO_SODR B`	`0x400E1030`	25
3	`PIO_SODR C`	`0x400E1230`	28
4	`PIO_SODR C`	`0x400E1230`	26
2	`PIO_CODR B`	`0x400E1034`	25
3	`PIO_CODR C`	`0x400E1234`	28
4	`PIO_CODR C`	`0x400E1234`	26

Au boulot…

Modifiez votre programme qui faisait clignoter la LED bleue pour remplacer `digitalWrite` par l'écriture des « bits qui vont bien » aux « endroits qui vont bien ».

Profitez-en pour faire aussi clignoter la LED verte, en opposition avec la bleue.

Indice : Application des décalages de bits et du transtypage.

Indice : Le type des entiers 32 bits est redéfini sous le nom `int32_t` pour éviter les ambiguïtés.

Au boulot. . .

Nous avons vu, dans une séance précédente, les champs de bits.

Pourquoi ne pas les utiliser pour décrire bit-à-bit les registres comme des champs de taille 1 d'un `unsigned int` ?

Au boulot...

À votre avis, quel est l'intérêt d'avoir des registres différents pour mettre une broche à 0V ou la mettre à 3.3V (ON ou OFF) ?

Intérêt d'une fonction plutôt que l'adressage direct

- Manipulation directe des registres :
  - Très rapide.
  - Mais très dépendant du MCU.
- Fournir une fonction dédiée dans une bibliothèque :
  - Pas nécessaire de lire la doc du MCU.
  - Permet une interface identique pour plusieurs MCU (cartes).
  - Pas nécessaire de lire les docs des MCUs.
  - Permet de faire d'autres vérifications pas faites ici.

- **Pointeurs sur fonctions et interruptions**

Au boulot. . .

En utilisant vos connaissances sur la lecture des broches, modifiez votre programme qui fait clignoter la (ou les) LED(s) pour que le clignotement débute uniquement lorsque le bouton le plus près des LEDs est pressé.

Testez plusieurs fois votre programme en le redémarrant avec le bouton *Reset* du `Arduino`. Remarquez-vous un comportement étrange ?

Au boulot. . .

Que faudrait-il faire pour que le clignotement s'arrête si l'autre bouton est pressé ?

Constat

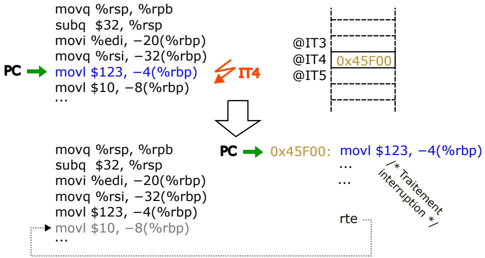
- On rate des évènements.
- Évènements asynchrones : on ne sait pas à l'avance quand ils arrivent.
- On passe notre temps à tester l'état des boutons.
- Les boutons sont très rarement appuyés.
- Problème courant sur n'importe quel ordinateur :
  - souris déplacée,
  - touche de clavier pressée,
  - données disponibles sur l'interface réseau,
  - donnée transférée $\leftrightarrow$ disque, ...

$\Rightarrow$ Nécessité d'un moyen pour exécuter du code de manière asynchrone.

$\Rightarrow$ Appel asynchrone de fonctions.

Interruption et gestionnaire d'interruption

- Gestionnaire : fonction accrochée à un évènement pour y réagir.
- Activer la détection de l'évènement recherché.
- Arrivée de l'évènement : déclenchement d'une interruption.
  1. Matériellement le CPU interrompt l'exécution du code courant.
  2. Va chercher, à une adresse fixe dépendante du type d'évènement, l'adresse du code (gestionnaire) à exécuter.
  3. $\Rightarrow$ Table d'adresses de fonctions (« vecteur »).
  4. Se déroute vers ce code.
  5. À la fin du traitement, retourne au code laissé en suspens.



Retrouver l'état mis en suspens

- Reprise du programme interrompu :
  - comment retrouver l'état des registres CPU ?
- Quelques registres CPU sauvegardés par le matériel.
- Sauvegarde des autres registres : à la charge du gestionnaire.
- Restauration par le gestionnaire des registres qu'il a sauvés.
- Restauration par le matériel des registrés qu'il a sauvés.

Mini au boulot : un gestionnaire d'interruption peut-il avoir des paramètres ?

Attacher un gestionnaire sur Arduino

- 3 types d'évènements sur les broches supportant des interruptions :
  - `RISING` : passage à l'état HIGH.
  - `FALLING` : passage à l'état LOW.
  - `CHANGE` : changement d'état (HIGH $\leftrightarrow$ LOW).
- `void attachInterrupt (uint32_t intno, void (*callback)(void), uint32_t mode)`
  - lier la fonction `callback`
  - à l'interruption numéro `intno`
  - lorsque l'évènement `mode` survient.
- Sur ce MCU, `intno` = numéro de broche.
- Plus portable : `uint32 digitalPinToInterrupt (uint32_t pin)`
- Modifie l'entrée du vecteur d'interruptions avec l'adresse du gestionnaire.
- + active l'interruption (+ un peu d'administratif).

$\Rightarrow$ Encore des manipulations de bits et d'adresses.

Au boulot...

Modifiez votre programme pour que le démarrage du clignotement soit déclenché par la détection par interruption de la pression de l'interrupteur.

Quel type d'évènement allez-vous surveiller ?

Qu'est-ce que cela implique sur votre fonction `loop` ?

Qu'est-ce que cela implique sur la variable `blink_started` ?

Variable volatile

- Programme fonctionne, mais...

- Variable `blink_started` modifiée par une fonction « non appelée ».
- Compilateur ne voit aucun appel syntaxique.
- Compilateur ne sait pas ce que fait `attachInterrupt`.
- Peut souhaiter optimiser `blink_started` qui semble constante.

⇒ La remplacer par `false`.

- Déclarer la variable `volatile` :
  - ▶ `volatile bool blink_started = false ;`
- Signifie : « variable peut changer à tout moment ».

⇒ Force le compilateur à relire sa valeur à chaque utilisation.

- Toujours pour des variables modifiées par des traitements en parallèle.
- Ici sans ça : on a eu de la chance et pas d'optimisations.

Au boulot. . .

Écrivez un programme qui compte le nombre de pressions effectuées sur le bouton (en utilisant les interruptions) et « affiche » ce nombre « en binaire » en utilisant les 3 LEDS présentes sur le montage.

Considérez que le LED bleue est le bit de poids fort.

Si vous le souhaitez vous pouvez également afficher la valeur de ce compteur dans le moniteur série.

Au boulot…

Proposez une solution pour adapter votre programme afin d'ignorer ces « bruits ».

Indication : il existe une fonction `unsigned long millis (void)` qui retourne le nombre de millisecondes écoulées depuis le dernier reset.

Implantez votre solution.

Au boulot. . .

Eh oui, les oscillations du relâchement provoquent aussi des fronts montants qui font croire à des pressions.

Proposez une solution pour régler ce problème et implantez-la.

Au boulot...

La fonction `millis ()` retourne un `unsigned long`.

Sur ce MCU, `unsigned long` = 32 bits.

Au bout de combien de temps notre compteur de temps va déborder ?

Au boulot. . .

Que va-t-il se passer si l'on laisse notre compteur déborder ?


Fin