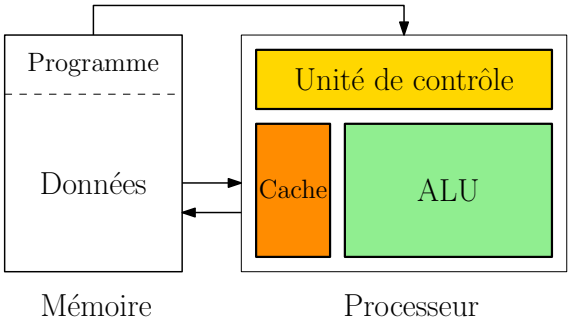
Programmation avancée en C :

Représentation des données

Licence informatique 3e année

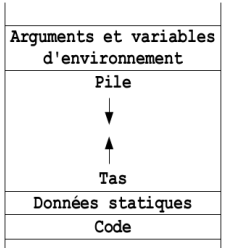
Université Gustave Eiffel

La machine



La mémoire

- Tout programme manipule de la mémoire.
- L'atome est l'**octet** = unité indivisible de 8 bits (chaque bit vaut 0 ou 1).
- **Pas de type (`int`, `float`, ...), que des bits !**
- L'art de coder en C : savoir modéliser/diviser/utiliser proprement et efficacement la mémoire pour encoder l'information nécessaire.
- Cette liberté est une force (efficacité) mais une complexité pour le programmeur (comprendre et savoir représenter).
- Positionnement de la mémoire utilisée par un programme :



Les types en langage C

- type = taille de zone mémoire + interprétation
- Quelques catégories : entiers, flottants, pointeurs
- Les entiers et les pointeurs : dépendant de la machine (et du système installé dessus...)
  - taille des pointeurs (c'est-à-dire des adresses) :
  - 4 octets sur une machine 32 bits
  - 8 octets sur une machine 64 bits
- L'opérateur `sizeof` pour la taille (en octet) d'un type
- Seule contrainte du langage C pour les tailles :

`sizeof(char)` ⩽ `sizeof(short)` ⩽ `sizeof(int)` ⩽ `sizeof(long)` = `sizeof(size_t)`

Les types entiers de base

type	taille	signé	valeurs assurées
signed char	1	oui	[-127, 127]
unsigned char	1	non	[0, 255]
char	1	Ça dépend du compilateur !	
(signed) int	4	oui	[-2 147 483 647, 2 147 483 647]
unsigned int	4	non	[0, 4 294 967 295]

- La norme C sur les types signés : entre $-(2^N - 1)$ et $2^N - 1$.
- Un compilateur **peut** autoriser -2^N (par exemple, un char valant -128).
  ⇒ Portabilité compromise *(Mais ça marchait chez moi...)*
- Représentations : décimale (`1234`), hexadécimale (`0x4D2`), octale (`02322`). Pas de binaire !

Débordement des types entiers

- Aucune vérification ! (force : plus rapide, faiblesse : n'attrape pas les erreurs)

```
1 unsigned char c = 255;
2 c = c + 1;
```

  ⇒ `c` vaut finalement 0.

- Cela peut provoquer des boucles infinies...

```
1 void count_down(int n){
2   unsigned int i;
3   for(i = n; i >= 0; i--){
4     printf("%d\n", i);
5   }
6 }
```


Opérateurs sur les entiers

- Les usuels : `9 + 4` (13), `9 - 4` (5), `9 * 4` (36)
- Quotient et reste de la division entière : `9 / 4` (2), `9 % 4` (1)
- Les raccourcis : `+=, -=, *=, /=, ...`
  `x += 4;` est le même que `x = x + 4;`
- Incrément et décrément : `i++`, `++i`, `i--`, `--i`
  - `i++` : utiliser la valeur, puis incrémenter
  - `++i` : incrémenter, puis utiliser la valeur (incrémentée)
  - Grande confusion possible : que vaut `i++ + ++i` ?
  - Solution : éviter les mélanges et les doubles usages !

Du bon usage des entiers

- On préférera utiliser le type `int` pour les entiers.
- Les types `char` et `short` sont utilisés uniquement pour économiser la mémoire, seulement quand c'est absolument nécessaire.
- Pour faire des opérations de manipulation de bits (on reverra cela plus tard), on utilisera de préférence `unsigned`.
- Pour stocker des valeurs positives ou nulles, on pourra utiliser `unsigned` si les opérations arithmétiques utilisées derrière restent simples.
- Si besoin, utiliser les types avec un nombre fixé de bit dans `stdint.h`.

Les types “réels” et leur représentations

- `float` : “réel” simple précision
- `double` : “réel” double précision
- Les représentations :
  - 12.34 ou 1234. ou .1234
  - notation scientifique : *mantisse* $\times 10^{exposant}$
    1723.68 = 1.72368e3 = 17.2368E2
    0.015 = 1.5e-2
  - par défaut, les constantes sont de type double
    `float` : 245.45f = 245.45F
    `double` : 245.45 = 245.45l = 245.45L
  - environ 7/15 chiffres significatifs pour les float / double

Problèmes de précision

- Calcul en virgule flottante ⇒ approximations
  (ne pas utiliser pour les calculs exacts, ex : finances)

```
1  int main(int argc, char* argv[]){
2    float f = 0.f;
3    double d = 0;
4    int i;
5    for(i=0; i<1000; i++) f = f + 0.1f;
6    for(i=0; i<10000000; i++) d = d + 0.1;
7    printf("f = %f\nd = %f\n", f, d);
8    return 0;
9  }
```

affiche au final

```
f = 99.999046
d = 1000000.000001
```

Ne pas comparer les flottants en utilisant == !

C’est pas rien deux fois rien !

L’addition des flottants n’est pas associative. A partir de là, toute l’arithmétique est à surveiller de près.

```
1  int main(int argc, char* argv[]){
2    double d = 67482746.433251;  /* un grand */
3    double rien = 0.000000249;   /* un petit */
4    printf("deux fois rien : %f\n", (d + rien) + rien);
5    printf("deux fois rien : %f\n", d + (rien + rien));
6    return 0;
7  }
```

affiche au final (oui je l’ai fait exprès, mais ça ne prévient pas quand ça bogue un programme...)

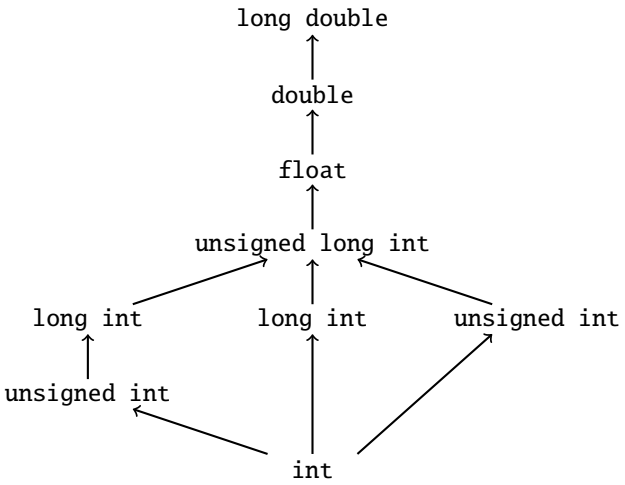
```
deux fois rien : 67482746.433252
deux fois rien : 67482746.433251
```


Opérations sur les réels

- Les usuels : `+, -, *`
- `/` : division réelle
- conversions automatiques : en général, entier vers flottant, moins précis vers plus précis
  `6 / 1.5` ⇒ `6.0 / 1.5`
  Exception : `int a=2.45;` ⇒ arrondi à `int a=2;` (cast implicit)
- Pour la division réelle de deux entiers, il faut une conversion explicite avant la division
  $6/5$ ⇒ `6.0 / 5`

```
int i = 2;
float f = 5 / (float)i;
```

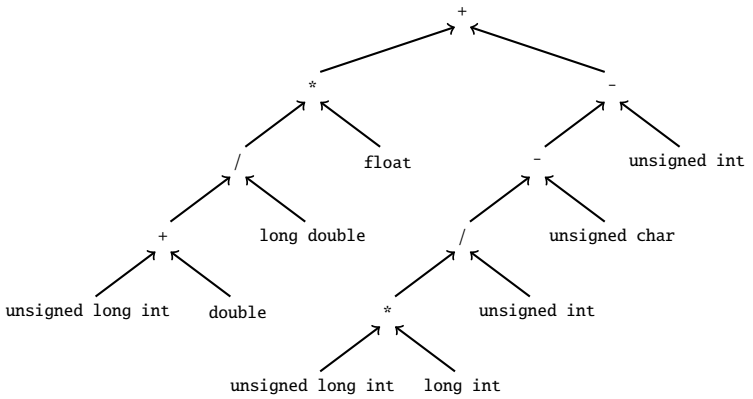
Conversions arithmétiques



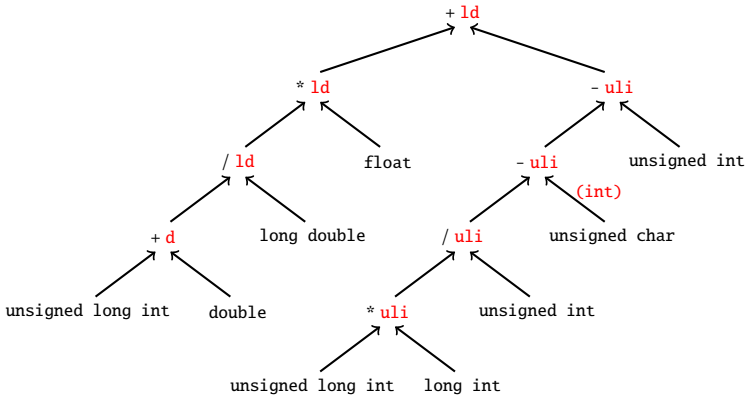
L'utilisation de `unsigned` peut rendre les conversions machines dépendantes lorsque `long int` contient `unsigned int`.

Exemple

Donner le type de :



Correction



Comportements machines dépendants

- arrondi de / pour les entiers : machine dépendant pour les opérandes négatives
- signe de % : machine dépendant pour les opérandes négatives
- division par 0 : comportement machine dépendant
- les dépassements de capacité

Le type caractère

- `char` s'agit d'un entier de 1 octet !
- On interprète les `char` comme des codes de caractères
  - pas de problème entre 0 et 127
  - au delà, dépend de l'encodage du système (à éviter)
- On désigne le code du caractère `x` par `'x'`
- Ne pas confondre caractère et valeur d'un chiffre (le char `'9'` n'est pas indexé par l'entier 9 dans la table ascii)
- On peut utiliser les opérateurs entiers
  - exemple : `'a'+1` vaut `'b'` (car les codes sont bien rangés)
  - aussi : `'0'+5` vaut `'5'`
- Et comme d'habitude, tout est dans `man ascii` !

Caractères spéciaux

- `'\n'` : saut de ligne
- `'\r'` : retour au début de ligne
- `'\t'` : tabulation (largeur variable)
- `'\\'`, `'\"'` et `'\''` pour les caractères `\`, `"` et `'`
- `'\0'` : fin d'une chaîne de caractères
- ... et les autres plus obscurs ...

Les types pointeur

- Pointeur = adresse (endroit) dans la mémoire
- Validité non assurée automatiquement !
- Déclaration : type*
  Exemple : `int*`, `float*`, `char*`, `char**`, `int****` ...
  Cas spécial : `void*`, qui est "sans interprétation"
- L'opérateur d'adresse : `&`
  Exemple : `&a` est l'adresse de la variable `a`
  Si `a` est un `int`, alors `&a` est de type `int*`
- L'opérateur d'indirection : `*`
  Exemple : Si `ptr` est de type `float*`, alors `*ptr` est l'espace de mémoire à l'adresse `ptr`, vu comme un flottant

Opération des pointeurs

- Décaler un pointeur : `+`
  Exemple : `int* ptr2 = ptr1 + 3;`, alors `ptr2` pointe vers le lieu situé 3 `int` après `ptr1`
- Le type signifie l'interprétation de la mémoire
  Exemple : `ptr + 3` est 12 octets après `ptr` s'il est du type `int*`, mais 3 octets après s'il est du type `char*` !
  Invalide pour `void*` !
- Raccourci d'indirection de décalage : `[]`
  Exemple : `ptr[i]` est équivalente à `*(ptr + i)`

Les tableaux

- Essentiellement un pointeur vers un espace de mémoire interprété comme des éléments contigus de même type
- Cet espace est réservé automatiquement par le programme.
- Un tableau est comme un pointeur, mais à un endroit fixé
- Déclaration : type `nom[taille];`
  `int t[N];`
  `double cosinus[360];`
- Avec initialisation :
  `char vowel[6] = {'a','e','i','o','u','y'};`
- La taille devient optionnelle :
  `char* name[] = {"Smith", "Doe", "X"};`

Les tableaux

- Numérotation de `0` à `taille-1`

```
1 int value[1024];
2 int i;
3 for (i = 0; i < 1024; i++){
4   value[i] = i;
5 }
```

- Raison : C'est un pointeur, donc premier élément est sans décalage (`value + 0`) !

Les erreurs sur les indices de tableaux restent la source de fautes de segmentation la plus courante en Licence... *Source : l'excité au tableau*

Les chaînes de caractères

- C'est juste un pointeur vers des caractères jusqu'à trouver `'\0'` (valeur ASCII 0)
- Délimitées par des guillemets
  `char msg[]="Welcome";`
- variantes :
  `char* msg="Welcome";`
  `char msg[]={'W','e','l','c','o','m','e','\0'};`
- Chaîne vide `""` ⇒ tableau dont la première case est `'\0'`
- Longueur = nombre de `char` avant `'\0'`
  `"abcd"` a une longueur de 4
- Concaténation automatique des chaînes constantes
  `"Gustave" "Eiffel"` est équivalent à `"GustaveEiffel"`

Les chaînes de caractères

- On ne peut ni modifier les chaînes statiques, ni les arguments de `main` !

```
1 int main(int argc, char* argv[]){
2   char* s="abcd";
3   s[2] = 'u';
4   printf("%s\n", s);
5   return 0;
6 }
```

BOOOUUUMMMMM !!!!!!!!!!

```
1 int main(int argc, char* argv[]){
2   argv[0][0] = '*';
3   printf("%s\n", argv[0]);
4   return 0;
5 }
```

BOOOUUUMMMMM !!!!!!!!!!

Taille

- Un tableau est un pointeur et ne sais pas la taille du tableau !
- l'opérateur `sizeof` ne fonctionne que pour les tableaux dont la taille est connue par la fonction à la compilation.

```
1  void test_sizeof(int t[]){
2    printf("sizeof(t) from function = %lu bytes\n", sizeof(t));
3  }
4
5  int main(int argc, char* argv[]){
6    int m[60];
7    test_sizeof(m);
8    printf("sizeof(m) from main = %lu bytes\n", sizeof(m));
9    return 0;
10 }
```

donne

```
sizeof(t) from function = 8 bytes
sizeof(m) from main = 240 bytes
```


Taille

3 solutions :

- Connaître la taille grâce à une constante
  `#define SIZE_TAB 256`
  `const int SIZE_TAB=256;`
  (la deuxième donne une constante visible par le débogueur)
- Passer la taille en paramètre
  `void affiche_tableau(int T[], int n);`
- Utiliser un élément marqueur comme `'\0'` pour les chaînes de caractères

Débordement

- Aucun contrôle de débordement (Ce sont des pointeurs !)

```
1  #define N 10
2
3  void foo(int A[], int B[]){
4    int i;
5    for (i = 0 ; i < 20 ; i++) B[i] = 33;
6    for (i = 0 ; i < N ; i++) printf("%d %d\n", A[i], B[i]);
7  }
8
9  int main(int argc, char* argv[]){
10   int B[N];
11   int A[N];
12   foo(A, B);
13   return 0;
14 }
```


Débordement

produit...

```
nborie@perceval:~> gcc -o test test2.c -Wall -ansi
nborie@perceval:~> ./test
33 33
33 33
33 33
33 33
33 33
33 33
33 33
33 33
1362109184 33
32767 33
```

deux 33 sont dans la nature... ou dans la ram

Tableaux à n dimensions

- `int t[100][16][45];`
- chaque `int t[i][j]` est un tableau de 45 `int`
- Ordre de parcours : C alloue les tableaux à deux dimensions par lignes (alors que `Fortran` le fait par colonnes)

  `int T[2][3];` donne en mémoire (@ l'adresse de `T[0][0]`) :

T[0][0]	T[0][1]	T[0][2]	T[1][0]	T[1][1]	T[1][2]
@+0	@+4	@+8	@+12	@+16	@+20

  ⇒ On boucle d'abord sur la première dimension !
- Quand on passe un tableau à une fonction, on doit mettre toutes les dimensions sauf la première.

  `void foo(int t[][]);` ⇒ Non !
  `void foo(int t[][M]);` ⇒ Ok !

Les variables

- Identificateurs : `[_a-zA-Z][_a-zA-Z0-9]*`
  (idem pour les noms de fonctions !)
  `Hello_89`, `__foo`, `b_l_a`, ~~`3tree`~~
- Éviter les noms proches des mots-clés : `new`, `class`...
- Appeler un chat un chat
  Un compteur peut facilement se déclarer
  `int compteur=0;`
  (code parlant = documentation allégée !!!)

  Donner des noms explicites aux variables et aux fonctions !!!

Les variables

- Doivent être déclarées avant d'être utilisées
- Déclaration : `int a;`
- Avec initialisation : `int a=3;`
- Déclarations multiples : `float f1=1.3, f2;`
- Attention :

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[]){
4   char* a="Hello", b="you";
5   printf("%s %s", a, b);
6   return 0;
7 }
```

  donne

```
initialization makes integer from pointer without a cast
```


Portée d'une variable

- Visible dans le bloc : A NE PAS FAIRE !!!

```
1 int main(int argc, char* argv[]){
2   int i;
3   for (i=0, i<10, i++){
4     int a = i;  /* Horrible !!! */
5   }
6   printf("a=%d\n", a);
7   return 0;
8 }
```

- Dans une fonction : variable locale
- Dans un fichier : variable globale (dans le même fichier)

Variable globales

- Utilisable dans tout le fichier
- Plutôt à éviter (nuit à la modularité, pas thread-safe)
- Dans quelques cas, on peut s'autoriser à des variables `static` sur un fichier

```
1  static int nb_echange = 0;
2
3  void tri_selection(...){
4    ...
5    nb_echange++;
6    ...
7  }
8
9  void tri_insertion(...){
10   ...
11   nb_echange++;
12   ...
13 }
```


Initialisation des variables

Il faut toujours initialiser les variables de façon que l'exécution ne dépende pas des valeurs par défaut (qui peut être n'importe quoi).

```
1  int sum(int n){
2    int i, x=0;
3    while (i<n){
4      x = x+1;
5      i++;
6    }
7    return x;
8  }
```

Le résultat dépend de la valeur par défaut de `i`.

```
1  int sum(int n){
2    int i, x=0;
3    for (i=0 ; i<n ; i++){
4      x = x+i;
5    }
6    return x;
7  }
```

`i` est ici initialisé avant utilisation par le `for`.

Affectation

- `=` : est un opérateur !
  `int a=3;` ⇒ déclare un entier `a` qui aura la valeur 3.
- L'opérateur `=` affecte une valeur à une variable mais a aussi une valeur de retour (qui est la valeur affectée)
  `int a=3;`
  `int b=(a=a+1)+3;` ⇒ `a` vaut 4 et `b` vaut 7
- Pratique pour tester les retours de fonction

```
1  if ((p = (int*)malloc(5*sizeof(int))) == NULL){
2    fprintf(stderr, "Erreur d'allocation\n");
3  }
```

  test propre de retour de `malloc`

- Ne pas abuser ! Ça pourrait rendre le code illisible ...

Effets de bord

- Modification d'une valeur pendant l'évaluation d'une fonction
- Dangereux
- Ne l'utiliser que dans des cas simples !

```
1  if ((foo(a) == 3) && (a++ == 0))
```

Horrible ! ! ! Quand est-ce que `a` est incrémenté ?

Conversion

- `unsigned char c = 713;`
  ⇒ (un des warning de `-Wall` s'active) attention : grand entier implicitement tronqué pour un type non signé [-Woverflow]
  ⇒ `c = 201`
- Ce qui se passe en binaire :

713	=	1	0	1	1	0	0	1	1
201	=			1	1	0	0	1	0	0	1
					⟵		8 bits		⟶		

- On évite le warning avec un cast :
  `variable = (type)expression`
- Changement d'interprétation d'un variable par cast :

```
float x = 3.1875f;
int ix = (int)x;
unsigned int bx = *(unsigned int*)(&x);
```

- Attention, il faut être sûr de ce que l'on fait avec cast.

Les constantes, type macro

- définies avec la commande :
  `#define IDENTIFICATEUR valeur`
- exemples :

```
#define PI 3.14
#define SIZE_MAX 1024
#define YES 'y'
#define PROMPT "$>"
```


Les constantes

- Remplacement brut de chaînes par le préprocesseur sauf à l'intérieur strict des identificateurs et dans les chaînes
- risques d'erreur

```
1  #define A 123
2
3  int main(int argc, char* argv[]){
4    int A = 12, Ab = 12;
5    printf("A = %d\n", A);
6    return 0;
7  }
```

donne après action du préprocesseur

```
1  int main(int argc, char* argv[]){
2    int 123 = 12, Ab = 12;
3    printf("A = %d\n", 123);
4    return 0;
5  }
```


Les constantes

- Toute constante non triviale doit être définie.
- Leur usage évite les modifications multiples.
- Donner des commentaires sur les constantes
- Bien faire attention aux collisions avec les autres identificateurs
- Convention : TOUS LES CONSTANTES EN MAJUSCULE