Résumé de l'aspect procédural du C et du C++

Sylvie Ratté et Hugues Saulnier

Septembre 1999

Service des enseignements généraux
École de technologie supérieure

Partie I
PRÉSENTATION GÉNÉRALE

1. CONCEPTS GÉNÉRAUX

Programmer en mode impératif, c'est donner à l'aide d'une syntaxe précise une suite d'ordres à une machine électronique. En C et en C++, la machine est virtuelle; la machine réelle est atteinte grâce au compilateur et au système d'exploitation.

Une **instruction** est l'équivalent code d'une action précise choisie parmi un groupe limité. Il s'agit donc d'une commande correcte, transformable en code machine par le compilateur. En C et C++ elle se termine « toujours » par un **point-virgule**.
Les langues impératives possèdent deux caractéristiques fondamentales :

- La présence de constructions spécifiant précisément l'ordre dans lequel les instructions doivent être exécutées;
- L'utilisation des effets de bord : la capacité de changer l'état de la machine souvent sans mention ou description explicite (ex. une lecture dans un fichier déplace le pointeur dans le fichier[1]).

On y privilégie le concept de programmation structurée dont le but ultime est de faire en sorte que la lecture du code s'apparente à la lecture du langage humain d'une part, et que le commentaire devienne presque inutile d'autre part (ce qui est loin d'être atteint...).

Toutes les implémentations de notre langue (Gnu, Borland, Symantec, Watcom, Microsoft, AT&T,...) possèdent un préprocesseur, un compilateur et un éditeur (ou bâtisseur) de liens (en anglais « linker »).

Le préprocesseur fait des tâches s'apparentant à celles d'un traitement de texte : le « remplacement » de mots ou de caractères (identifiés par #define), l'élimination des commentaires (identifiés par « /* » suivi de « */ ») ou du code « indésirable » (identifié par #if, #elif, #else, #endif, #ifdef ou #ifndef) ou l'insertion de fichiers (identifiée par #include).

Le compilateur construit ensuite, à partir de ce code « simplifié », un fichier de commandes machine propres à la machine hôte (sous forme « obj » standard). Finalement l'éditeur de lien, comme son nom l'indique, effectue les liens entre le code du fichier « obj » obtenu par la compilation du programme et le code « obj » des librairies possiblement utilisées par le programme (libraries qui peuvent être personnelles ou standards) et produire ainsi le fichier pouvant être exécuté sur la machine hôte.

[1] Il s'agit bien d'un effet de bord puisque ce déplacement n'est pas mentionné explicitement par l'instruction de lecture.

A. Commentaires et directives

L'introduction de commentaires se fait aisément : tout ce qu'on retrouve entre les séquences « /* » et « */ ».

Il vous est aussi possible d'adopter les commentaires offerts par le C++ : tout ce qui suit « // » sur une ligne est un commentaire.

Tous ces ajouts sont évacués du source par le préprocesseur avant le passage dans le compilateur.

Toute ligne dans un fichier commençant par le caractère dièse, « # », est spéciale, elle identifie une **directive** au préprocesseur. Ces lignes ne concernent strictement que le préprocesseur (et non le compilateur). Conséquemment, elles ne sont pas incluses dans ce que l'on appelle « le code ». L'ensemble des directives possibles est présenté à l'annexe A.

B. Composition d'un fichier C ou C++

Un programme en langage C ou C++ est constitué d'un ou plusieurs fichiers. La technique du multi-fichiers à compilation séparée, tout en étant relativement simple, nécessite le travail commun du préprocesseur et de l'éditeur de liens.

Ces fichiers ne contiennent que des **identificateurs**, des **opérateurs**, des **séparateurs** et des **littéraux** qui forment ce qu'on appelle le code. On y retrouve aussi les **directives** au préprocesseur.

La notion de bloc

Le bloc est la structure de base de la langue. Un bloc regroupe plusieurs instructions simples. On le reconnaît à ses **accolades** de début et de fin.

Les blocs sont **imbricables** à volonté, mais les **blocs** les plus **externes** possèdent **toujours** un **nom** (ce sont des **fonctions**) et sont donc très différents des blocs internes.

Tout programme correct possède une **fonction** nommée **main**, dont le code s'exécute au départ.
Et voici le plus petit programme possible :

```
int main( ) { return 0 ; }
```

C. Notion de type

Un type, c'est un ensemble de valeurs et d'opérations permises sur ces valeurs.

Pour éviter « autant que possible » les erreurs de programmation, notre langue s'appuie sur un système de types (en anglais « type system »). Il s'agit en fait d'un ensemble de règles qui associe un **type** à toute évaluation.
Une expression dans votre code à laquelle le système de types (ST) ne peut associer un type sera rejetée par le compilateur. On dit d'ailleurs du C ANSI qu'il possède un système de types robuste (« strong type system ») mais que celui du C++ l'est encore davantage. Il existe cependant des langages encore plus pointilleux au niveau des types.

Pendant l'exécution, les **données** utilisées auxquelles on donne le vocable **d'objet** possèdent une **place** dans la mémoire électronique où un **train de bits** représente chacune d'elles. Certaines possèdent un **nom**, grâce à l'identificateur qui leur est associé, d'autres sont anonymes. Mais chaque **objet** a un **type bien défini** qui **détermine** les **opérations** que votre code pourra lui faire subir.

Un programmeur peut tenter de modifier le type d'une expression qui possède déjà un type. En informatique, on nomme cette action « **type cast »** (en français, on dit « coercition de type » ou « forçage de type »). Dépendant du contexte, le compilateur est libre ou forcé d'accorder cette faveur au programmeur.

La syntaxe en est simple : si A est de type T1 et que T2 est un type alors

```
( T2 ) A
```

est une action de « coercition de type » et fourni un objet de type T2 si le compilateur ne génère pas d'erreurs.

Types de base

Toutes les machines possèdent un jeu de caractères, des nombres entiers, des nombres réels et leurs opérations respectives. On interagit avec ces ensembles grâce à ce qu'on appelle les **types de base** (tout un groupe de mots réservés) : **char** pour les caractères, **int** pour les entiers et **double** pour les réels.

De façon surprenante, la **valeur de vérité** (le vrai ou faux logique) est représentée numériquement : zéro est l'unique représentant du faux et toutes les autres valeurs représentent le vrai.

Puisque nous travaillons avec une représentation finie, la capacité d'un **type de base** dépend de la machine et du compilateur utilisé: sur un PC les **char** sont codés sur un **octet**, les **int** sur deux ou quatre et les **double** sur huit.
Pour faire bonne mesure, il existe également des **modificateurs,** les préfixes **long**, **short signed** et **unsigned**, permettant d'obtenir des variantes sur ces types de base**.** Ainsi le préfixe **long** ajouté à un type de base vous assure que le nouveau type est au moins aussi étendu que le type sans préfixe (prenez le dual pour **short**). Le préfixe **unsigned** sur un type entier, qui économisant le bit du signe, vous permet de doubler la capacité en positifs.

Par exemple dans le monde des compilateurs de la compagnie Borland:

- un **short double** (mieux connu comme **float)** vit sur quatre octets,
- un **long int** (mieux connu comme **long)** vit sur quatre octets,
- un **long double** vit sur dix octets.

L'**opérateur sizeof** qui retourne le nombre d'octets utilisés par un objet ou par un type, vous permettra de vérifier aisément la taille de tous ces types offerts. Par exemple : **sizeof (int) , sizeof (2.32+4) , sizeof (double) , sizeof (1)** sont des expressions entières correctes.

Littéraux

On appelle **littéraux**, les valeurs « écrites » (non calculées) appartenant aux **types de base**:

- **1234** un **int** ,
- **12.45633** un **double**,
- **0356L** un **long int** en base 8 (en **octal**),
- **0x2A3F5LU** un **unsigned long int** donné en base 16.
- **0x2C3** un **int** donné en base 16
- **2.45f** un **float**,
- **34.5672L** un **long double**,

Apparaissent ici des **préfixes** (0,0x) et des **suffixes** (L,f,U). Ces façons de construire sont usuelles, vous devez vous y habituer rapidement et avoir l'oeil aux détails.

Les suites de caractères entre guillemets comme "dommage" sont automatiquement reconnues comme des suites de **char** !

Promotion

Il existe une hiérarchie naturelle à ces types de base: les entiers peuvent être écrits comme des réels sans perte de précision alors que l'inverse n'est pas vrai. Cette situation est même possible à l'intérieur de la famille des entiers: entre **int** et **long int** par exemple. On retrouve donc un système automatique mais souvent dangereux de promotion de types. Ainsi le produit d'un **int** et d'un **double** sera typé **double**. L'annexe B présente un tableau complet des conversions possibles.

2. IDENTIFICATEURS

A. Règles syntaxiques

Tous les identificateurs suivent les mêmes règles :

- On distingue majuscules et minuscules.
- Le premier caractère est une lettre ou le caractère souligné '_' et la suite ne contient que des lettres, des chiffres ou le caractère souligné '_'.
- Il existe de plus une règle tacite chez les programmeurs : ne jamais utiliser un identificateur dont le nom commence par le caractère souligné '_' car ces mots spéciaux sont souvent utilisés par la compagnie mère du compilateur utilisé.
- Un petit nombre d'identificateurs forment la base du langage et toute tentative pour les redéfinir est interdite. Ce groupe particulier porte le nom de **mots réservés**. L'annexe C et l'annexe D présentent la liste des mots réservés du C et du C++.

B. Déclaration

Une déclaration introduit un nouvel identificateur.

C'est l'unique façon d'introduire un nouveau nom dans un programme et les déclarations suivent des règles très strictes. En C, une déclaration peut apparaître au début d'un bloc avant les instructions qu'on y trouve ou hors de tout bloc.

Une déclaration peut également apparaître n'importe où dans un bloc.

Le principe général est simple : Pas d'utilisation avant déclaration!

Sans entrer dans les détails sérieux, vous pouvez déclarer :

- un **type** : appartenant au programmeur grâce à **struct** , **enum** , **union** ou **typedef**,
- une **fonction** : le nom d'un bloc externe que vous voulez coder,
- une **variable** : un nom pour un objet que vous utilisez.

L'**accessibilité** d'un identificateur correspond à la partie du programme où il est utilisable.

C. Classe de stockage ou d'allocation

À toute **déclaration** est associée une classe d'allocation (en anglais « **storage class** ») qui permet d'obtenir certaines caractéristiques spéciales concernant son implémentation : sa durée de vie, son emplacement relatif en mémoire, son accessibilité. Le concept de « classe d'allocation » est relativement complexe et s'appuie sur les préfixes **auto, register, extern, static.** En voici de simples descriptions :

- **auto** est uniquement permis lors de déclarations de variables locales, donc à l'intérieur d'une fonction (n'oubliez pas que « main » est aussi une fonction). Comme c'est la classe d'allocation par défaut, il est rarement écrit.
- **register** est uniquement permis lors de la déclaration de variables locales. On exprime par là le **souhait** d'un accès rapide à la variable.
- **extern** sert (surtout en programmation multi-fichiers) à indiquer que l'identificateur qu'il précède a déjà eu une allocation mémoire sous le même nom. On dit alors qu'il y a eu **définition de la variable**. L'éditeur de liens a alors le devoir de rechercher le fichier où la définition de l'identificateur se réalise et de faire en sorte que l'utilisation de l'identificateur ainsi mentionné réfère à cette position mémoire. Le préfixe est donc **explicite** dans la déclaration de l'identificateur du fichier utilisateur. Les fonctions appartiennent **par défaut** à cette classe d'allocation. Malgré tout, bien des programmeurs tiennent à le rendent visible et l'écrivent. La norme reste neutre quand à l'utilisation du préfixe dans la déclaration générant la définition.
- **static**, dans la déclaration d'une fonction ou d'une variable déclarée hors de tout bloc, sert à limiter l'utilisation du nouvel identificateur au seul fichier où il se trouve. Elle devient alors (par abus de langage) privée. Dans un bloc, la déclaration d'une variable **static** lui donne une durée de vie égale à celle du programme tout en limitant son accessibilité au bloc où elle est déclarée.

D. Les identificateurs d'objets (ou de données)

Variables

Chaque variable possède un **nom**, un **type** et une **place** bien définie en mémoire (grâce à sa classe d'allocation).

Une variable déclarée hors de tout bloc est dite globale et est initialisé à zéro par défaut. Sa durée de vie est alors égale à celle du programme et elle est accessible dans tous les blocs, à moins qu'un bloc possède une variable homonyme. Nous vous demandons **très fortement** de les éviter **à** moins qu'elle ne soit préfixée de **static** dans une librairie personnelle[2]. Une variable locale **auto** ou **register** n'est pas initialisée par défaut. La déclaration d'une variable respecte la syntaxe suivante :

```
type nom [ = valeur ];
```

Le *type* est un type de base ou un type défini par le programmeur. Le type spécifie la taille de l'emplacement mémoire à réserver et les opérations permises sur cet emplacement.
La partie **[=valeur],** qu'on appelle l'initialisation (ou définition immédiate), est **facultative**.
Par exemple :

```
double radis = 2.8; int koto; char carac = 's';
```

sont des instructions correctes. Donc *radis* vaut 2.8 et la valeur de *koto* est imprévisible si l'instruction apparaît dans un bloc...

La **durée de vie** des variables automatiques est égale au temps d'exécution entre le moment de la **définition** de la variable et la fin du **bloc**.

Une instruction comme :

```
ma_variable = cette_valeur ;
```

s'appelle une **assignation**. On traduirait en français par : « dans l'espace mémoire connu sous le nom de *ma_variable* met *cette_valeur* ». On dit alors que *ma_variable* est une **lvalue,** pour « leftvalue » c'est-à-dire un objet pouvant être mis à gauche d'une assignation.

Si vous êtes curieux, on peut facilement **obtenir l'adresse** en mémoire d'une **lvalue** avec **l'opérateur &** utilisé en préfixe. Ains l'expression *&ma_variable* me donne l'adresse en mémoire de l'emplacement réservé à *ma_variable*. Cet opérateur ne doit cependant pas être appliqué à un objet littéral ou encore au résultat d'une évaluation (sauf quelques exceptions). Dès lors, si *A* et *B* sont deux variables de type **double**, l'expression &(A+B) n'est pas valable.

Tant que vous respectez les limites imposées par le type de vos variables, vous êtes libre d'y placer les valeurs que vous voulez.

[2] Leur emploi doit cependant être très bien justifié puisqu'il ne faut pas oublier qu'elles sont tout de même des variables globales et que, de ce fait, elles ont tendance à rendre le programme difficile à lire.

Constantes

Elles suivent des règles très semblables à celles des variables. On obtient une constante en préfixant la déclaration d'une variable du mot réservé **const**. Cette déclaration nécessite obligatoirement la partie « =valeur ». Les constantes possèdent donc un **nom**, un **type**, une **place** en mémoire et une **valeur**.

```
const type nom = valeur ;
```

La **durée de vie** et l'**accessibilité** des constantes sont celles des variables.

Toute tentative pour changer cette valeur ultérieurement dans le programme sera détectée par le compilateur et refusée. Étant donné la stabilité qui en découle, on accepte très bien que des constantes soient définies hors de tout bloc c'est-à-dire qu'elles soient globales.

Voici deux petits exemples de constante :

```
const double constante_du_sirop = 1.234000347;
const long int voltage = 220;
```

Le concept de constante a pris une importance capitale en C++.

Alias (ou références)

Les alias n'existent qu'en C++. Avec la déclaration suivante :

```
type & nom = variable ;
```

on obtient ainsi un nouveau ***nom*** pour une variable **déjà existante** d'où leur titre d'alias. Il est important de noter que cet alias ne se voit pas alloué de **place** en mémoire puisque la variable en possède déjà une.

Les alias doivent obligatoirement être initialisés à une variable et **changer ultérieurement ce lien est impossible**.

Les alias seront très importants lors des passages de paramètres aux fonctions en C++.

Pointeurs et tableaux

Le **tableau** constitue la généralisation du concept de variable. Comme nous le verrons plus en détails à la section 2, un tableau est en fait une suite de **n variables** contigües du même type. Le **pointeur** est une type un peu spécial. Il s'agit d'une variable contenant l'adresse mémoire bien typée d'un autre emplacement en mémoire. Nous examinons évalement ce concept à la section 2.

Rôle des librairies[3]

Le C et le C++ ne possèdent qu'une base minimale; ils ne contiennent même pas de fonctions d'entrée-sortie élémentaires. C'est à travers les librairies standards qu'on obtient toute la puissance de la langue (le comportement des fonctions qu'on y trouve est décrit dans la norme ANSI et un fournisseur de compilateur a l'obligation de s'y conformer pour avoir droit au sigle ANSI). Leurs fichiers **interfaces** (stdlib.h, stdio.h, math.h...) doivent être **explicitement** inclus dans vos programmes (grâce à la directive #include). Ces fichiers vous donnent ainsi accès aux **identificateurs** offerts par la librairie.

E. L'exécution

Le cours normal d'exécution du code se résume facilement: **après une instruction c'est l'instruction suivante qui est exécutée**. Évidemment tout programme pensé par un humain contient des phrases comme « si l'on est dans cet état alors on doit faire cela » ou « répète ceci jusqu'à ce que cette condition soit vraie ». Ces formes une fois codées brisent le cours normal d'exécution : la première fait en sorte que parfois du code n'est pas exécuté tandis que la seconde fait en sorte que du code est exécuté un certain nombre de fois.

La commande **goto...***label* (l'infâme goto) permet un saut inconditionnel à l'intérieur d'une fonction. Elle brise de façon brutale le cours normal d'exécution et fait en sorte qu'un programme n'a plus, à la lecture, de logique apparente. La programmation moderne tente de les éviter comme la peste!! C'est tout simple, nous vous interdisons formellement le **goto** dans ce cours.

Structures de contrôle

Les formes acceptables de bris du cours normal d'exécution sont quasi toutes regroupées ici et portent le nom de **structures de contrôle**.

LE IF

permet d'exécuter un bloc d'instructions en fonction d'une certaine condition.

```
if (condition) bloc1
```

Vous remarquez que la **condition doit être entre parenthèses** et doit pouvoir être interprétée comme une valeur de vérité (numériquement). Le **if** terminé, on retourne au cours normal d'exécution.

LE IF…ELSE

permet d'exécuter telle ou telle portion du programme, en fonction d'une certaine **condition**.

```
if (condition) bloc1 else bloc2
```

[3] La norme ANSI mentionnée dans le texte réfère exclusivement au langage C.

Si la **condition** est différente de zéro, **bloc1** est exécuté sinon **bloc2** est exécuté. Le **if...else** terminé, on retourne au cours normal d'exécution.

LE SWITCH

est une instruction **bizarre** qui passe malheureusement souvent pour un simple test à choix multiple mais attention ce n'est pas vrai!

```
switch (expression) blocA
```

Dans **blocA**, sont définies un groupe de portes d'entrée (par **case** *constante_i***:** ou **default:**) et un groupe de portes de sortie (par **break**). L'ordre d'apparition de ces portes est totalement libre (à la façon des labels de gotos).

Une fois entré dans **blocA** , tout le code jusqu'au premier **break** ou jusqu'à la fin du bloc est exécuté. Une fois sorti de **blocA**, on retourne au cours normal d'exécution.

LA BOUCLE WHILE

permet d'exécuter du code un certain nombre de fois.

```
while (condition)
  blocA
```

La condition est testée, si elle est vraie, blocA est exécuté et on recommence jusqu'à ce que la condition soit fausse. Lorsque cela se produit, on retourne au cours normal d'exécution.

LA BOUCLE DO...WHILE

permet également d'exécuter du code un certain nombre de fois.

```
do
  blocA
while (condition);
```

Elle diffère du **while** de façon subtile: dans le **do ... while** on exécute **blocA** puis on teste si la **condition** est vraie et on recommence jusqu'à ce que la condition soit fausse. La boucle **do ... while** est donc toujours exécutée au moins une fois. Lorsque la condition devient fausse, on retourne évidemment au cours normal d'exécution.

LA BOUCLE FOR

permet également d'exécuter du code un certain nombre de fois.

```
for (instructions1 ; condition ; instructions2 )
  blocA
```

se comprend bien en réécrivant la séquence sous-jacente avec le **while**:

```
instructions1 ;
while (condition)
{
    blocA ;
    instructions2 ;
}
```

On dit que le **for** possède trois champs: l'initialisation (instructions1), la condition et l'incrément (instructions2).

Il est très facile de rendre les boucles *for* pénibles à déchiffrer. C'est pourquoi, parmi les programmeurs civilisés, on admet quelques principes de bonne programmation dont quelques-uns concernent cette boucle : on doit, dans la mesure du possible, éviter d'utiliser l'opérateur virgule à l'intérieur des trois champs; on ne place qu'une seule condition dans le second champ; finalement, le troisième champ ne devrait servir qu'à incrémenter le compteur lorsque l'on connaît le nombre de répétitions.

LA COMMANDE BREAK

n'apparaît que dans une boucle ou un **switch** (déjà vu). Elle provoque la fin de la boucle qui la contient et permet donc de terminer une boucle sans que la condition d'arrêt officielle soit remplie. Le break retourne au cours normal d'exécution.

Vous verrez d'ailleurs fréquemment des boucles pseudo-infinies telles :

```
While (1) bloc1
for ( ; ; ) bloc1
```

avec dans **bloc1** une instruction de forme :

```
if ( cette_condition ) break;
```

qui permet, lorsque *cette_condition* est vraie, de terminer la boucle qui semblait infinie. Cette forme de sortie de boucle est souvent plus naturelle que celle qui consiste à réorganiser le code afin de ramener la mise à jour de la condition à la fin de la boucle. On ne doit cependant jamais en abuser : votre code doit être bien structuré et on ne devrait pas trouver plus d'un **break** dans le corps d'une boucle (sinon retournez en forme classique : lorque le champ condition est faux, on cesse de boucler).

LA COMMANDE CONTINUE

Cette instruction provoque le « rebouclage » immédiat de la boucle **do**, **while** ou **for** qui contient le **continue**. Quand on exécute un **continue** on fait comme si l'on venait de finir d'exécuter la dernière des instructions du corps de la boucle (on est donc sur le point de « reboucler »). Dans le cas d'un **for**, si l'on rencontre une instruction **continue,** on évalue alors immédiatement le troisième champ, **instructions2**. Les limites d'utilisation du **break** s'appliquent également au **continue**.

F. Les fonctions

Un **bloc externe de code** doit posséder un **nom** et peut être exécuté en utilisant correctement son nom dans une instruction. Ces bloc nommés sont en fait des fonctions.

Que ces fonctions soient **définies** dans le programme ou que leurs déclarations soient **importées** des librairies classiques (stdlib.h, stdio.h, math.h...) ou de librairies personnelles, importe peu : un **appel** correct (une bonne utilisation de la fonction dans une instruction) accepté par le système de types du compilateur, force la machine à exécuter le code connu sous ce **nom**.

Syntaxe

```
TypeAuRetour   nom ( liste de paramètres )
```

TypeAuRetour indique le **type** de **la valeur** qui **prendra littéralement la place de l'appel** dans l'instruction utilisant la fonction. Cette **valeur** est explicitement et obligatoirement retournée par le code de la fonction grâce au mot réservé **return**. Les fonctions sont des assistants souples où l'on retrouve toujours le concept du « c'est fini! donne-moi le résultat ». On voit donc souvent dans leur code plusieurs **return** : chacun correspondant à certaines conditions d'utilisation de la fonction. Si les conditions d'un **return** sont satisfaites, le résultat est retourné et le reste du code de la fonction n'est évidemment pas exécuté[4]. Le cours normal d'exécution ne semble donc pas troublé par l'appel d'une fonction.

Les deux **parenthèses** à la suite du **nom** sont **obligatoires** : on **différencie** la déclaration d'une fonction des autres déclarations (des variables par exemple) grâce à elles. Elles contiennent les **paramètres** formels (arguments) de la fonction. Ce sont en fait de simples **variables locales (auto** ou **register)** au bloc de la fonction. Les paramètres formels **sont initialisés** par les objets présents à l'intérieur des parenthèses lors de l'**appel** de la fonction (les paramètres actuels). Cette initialisation automatique vous est offerte gracieusement par le compilateur.

Mais attention, cette merveilleuse simplicité cache de troublantes réalités informatiques. Vue du côté matériel, l'appel d'une fonction dans un programme « ressemble » à l'exécution d'une application par le système d'exploitation : sauvegarde de l'environnement actuel (où en est-on dans l'exécution du code, quel est l'état des registres du CPU, détails sur l'échange d'informations, etc.), pour ensuite passer la main au sous-programme le temps de son exécution : la fonction s'installe et prend le contrôle de la machine le temps d'exécuter ses commandes puis l'environnement antérieur se réinstalle. Tout cela demande bien du temps et de la place... On parle alors d'« overhead » provoqué par tout appel à une fonction.

En C classique, le C antérieur à la norme ANSI, les paramètres actuels de l'appel d'une fonction étaient littéralement transportés dans la « pile » sans tenir aucun compte du type des objets locaux prévus pour leur récupération. La moindre incohérence dans les types utilisés pour les récupérer générait un décalage catastrophique : des valeurs imprévisibles étaient récupérées lors de tous les

[4] On peut évidemment troubler cette douce tranquilité ordonnée en produisant des effets de bord pervers sur des variables globales d'où l'interdiction absolue de les utiliser dans ce cours.

transferts subséquents. Aucun mécanisme automatique de contrôle des types n'était prévu. Le risque incroyable d'erreurs amena la création d'une norme et à son respect par les programmeurs. La norme ANSI et le C++ vous offre donc la **vérification** du type des arguments à l'appel d'une fonction, leur conversion possible en cas d'inégalité mais de compatibilité et des passages le plus souvent sécuritaires.

Prototypes

On sépare la **déclaration** de la **définition** des fonctions. La déclaration d'une fonction est appelée **prototype**. Elle n'est pas obligatoire dans les petits programmes tenant dans un seul fichier mais la programmation moderne se fait le plus souvent en multi-fichiers à compilation séparée et l'utilisation des prototypes y est obligatoire.

Le prototype d'une fonction peut facilement être différencié de sa définition :

```
TypeAuRetour nom (liste de paramètres ) ;
```

Le prototype se **termine** par un point-virgule et le **bloc** de code qui **définit** la fonction **n'est pas** présent. On ne retrouve normalement entre parenthèses que les types sans les noms des paramètres (en quelque sorte, prototype veut dire « pour les types ») et c'est grâce à cette liste que le contrôle automatique des types s'effectuera.

À propos du C++

Le C++ vous offre en plus deux caractéristiques de très haut niveau pour les fonctions : la surcharge et les patrons (en anglais « template »).

SURCHARGE

Cette notion implique que plusieurs fonctions peuvent porter le même nom en autant que leur liste de paramètres diffèrent. En voici deux exemples simples avec leurs prototypes :

EXEMPLE 1 : La première fonction reçoit le nom d'un tableau d'entiers et sa taille. Elle retourne l'indice (la position) du plus petit élément. La seconde reçoit le nom d'un tableau d'entiers et deux positions. Elle retourne l'indice du plus petit élément entre ces deux positions reçues.

```
int indic_min( int [] , int ) ;
int indic_min( int [] , int , int ) ;
```

EXEMPLE 2 : La première fonction prédicat reçoit un entier et retourne 1 si le nombre reçu est premier, 0 sinon. La seconde reçoit un entier et l'alias d'un entier pour retourner 1 si le nombre reçu est premier, 0 sinon. S'il n'est pas premier l'alias contient le premier diviseur trouvé.

```
int premier( long int ) ;
int premier( long int , long int & ) ;
```

Évidemment, il est préférable d'utiliser cette **faveur** informatique uniquement dans les cas où les fonctions sont intimement reliées et que cela ne mène pas à des ambiguïtés de types. Prenons les deux fonctions « toto » suivantes :

```
void toto( int , double ) ;
void toto( double , int ) ;
```

Ces fonctions sont différentes pour le compilateur et il est concevable de penser que leur traitement diffère puisque le programmeur a jugé bon d'en mettre deux. Mais que se passe-t-il à l'appel suivant :

```
toto ( 12 , 35)
```

Oups! N'oubliez pas qu'il existe un mécanisme de promotion automatique des types et que celui-ci s'applique. Une des deux fonctions sera appelée mais laquelle?!

PATRON DE FONCTIONS (OU « TEMPLATE »)

Il existe des cas où le traitement offert par nos fonctions est tellement général qu'il pourrait facilement s'appliquer à n'importe quel type. Ainsi la fonction **indic_min** précédente ne peut travailler que sur des tableaux d'entiers. On conviendra que s'il s'agissait d'un tableau de réels, on obtiendrait le même algorithme; la seule différence serait située au niveau des types mentionnés dans le code (double vs int). Le C++ vous offre gratuitement le concept « **pour un type à venir** ». La forme « template » de la fonction « indic_min » devient (« T » ici n'est qu,un identificateur que vous choisissez) :

```
template <class T>
int indic_min ( T [ ] , int ) ;
```

Il s'agit véritablement d'un patron. Si on appelle cette fonction, dans notre programme, avec un tableau de « int » puis avec un tableau de « double », le compilateur C++ génère deux version de la fonction. Le compilateur génèrera donc autant de versions qu'il existe d'appels (dans votre programme) avec des types distincts. Si, par contre, la fonction n'est jamais appelée par votre programme, la fonction n'est jamais définie.

On notera qu'une fonction en « template » perd le privilège offert par le système de promotion automatique des types.

Partie II
TABLEAUX ET POINTEURS EN C

1. LA NOTION D'ADRESSE

A. Introduction

La mémoire d'un ordinateur peut être considérée comme une structure linéaire d'octets. Chaque octet peut être identifié par un indice entier positif, son **adresse**. Deux octets contigus ont des adresses différant de 1.

Un programmeur peut se servir convenablement d'un ou de plusieurs octets à partir d'une adresse. Un type « adresse » est représenté par un type connu suivi du caractère *. Par exemple **int *** ou **double *** sont des types « adresse » et, comme tout autre type, peuvent servir lors d'une coercition de type. À la limite, le programmeur peut aussi faire de l'adressage absolu. Ainsi, l'expression :

```
(int *) 25678
```

est correcte et permet au programmeur de lire ou d'écrire un entier à partir de l'octet 25678 (avec tous les dangers que cela comporte).

Le nom d'une fonction utilisé dans une expression est automatiquement évalué comme étant **l'adresse du premier octet de son code**. Ce type d'adresse est d'emploi très restrictif pour le programmeur. De toute façon, dans le cadre de ce cours, nous aborderons à peine ce concept.

Accès aux adresses

Deux opérateurs (* et &) sont spécialisés dans l'utilisation correcte des adresses.

1. L'opérateur d'adressage « & » placé en préfixe à une *Lvalue* de type T, permet d'obtenir l'adresse de la *Lvalue*. Notons que cette adresse est *bien typée*, elle est de type « T * ».
2. L'opérateur d'indirection « * » placé en préfixe à une adresse *bien typée*, permet d'obtenir, dépendant du contexte d'utilisation, une *Rvalue* (la valeur bien typée à cette adresse) ou une *Lvalue* (la capacité de modifier l'objet situé à cette adresse).

B. Arithmétique sur les adresses

Les adresses sont entières et certaines opérations arithmétiques leur sont accessibles. Ainsi, l'adresse d'un double sur 8 octets (notons la *riri*) à laquelle on ajoute 2 grâce à l'expression « riri+2 » est évaluée comme « riri+2*sizeof(double) », qui représente l'adresse d'un double situé 16 octets plus loin en mémoire. On peut ainsi additionner un entier à une adresse, soustraire deux

adresses de même type, incrémenter, décrémenter. Évidemment, le produit ou le quotient ou le modulo d'adresse n'ont aucun sens et sont interdits.

Les opérateurs de comparaisons, égal, inférieur à, supérieur à, sont aussi permis mais uniquement sur des adresses de mêmes types. Sinon, une adresse pourrait en égaler une autre mais ce ne serait plus nécessairement vrai en additionnant 1 aux deux adresses ! ?

Mais à la vérité, comme vous n'avez aucune responsabilité quant à la place qu'occupent dans la machine vos variables, vos constantes, vos fonctions et les valeurs prises par vos expressions, l'arithmétique ou la comparaison d'adresses n'ont vraiment de sens qu'à l'intérieur d'un tableau.

2. TABLEAUX ET POINTEURS

A. Tableaux

Si vous ajoutez un entier positif entre crochets en suffixe à la déclaration d'une variable :

```
Type  nom [ n ]
```

vous obtenez un **tableau** c'est-à-dire une suite de **n** variables du même type. Ainsi, « **int tab[100];** » nous donne le nom **tab** (vous verrez comme il est spécial) et une suite de 100 entiers sur **100*sizeof(int)** octets contigus. Déclarer des tableaux est possible autant avec les types de base qu'avec vos propres types et leur durée de vie et leur accessibilité sont celles des variables ordinaires.

tab[k] servira de **nom** à la **(k+1)^{ième} variable du tableau tab**. Le premier emplacement du tableau porte toujours l'indice zéro. Ainsi, **tab[0]** donne accès au premier élément, **tab[5]** au sixième et si **tab** a **n éléments , tab[n]** est à **l'extérieur** du tableau! Donc attention! Si le tableau **tab** possède **n** éléments, logiquement dans l'expression « **tab[k]** », **k** doit appartenir à l'intervalle **[0, (n-1)].** Le compilateur **n'interdit pas** l'utilisation d'un **k** quelconque!! **Modifier** par erreur un emplacement hors du tableau a toujours des **effets catastrophiques** et **consulter** hors du tableau donne des **résultats** pour le moins **surprenants**. Mais quelques programmeurs retors et expérimentés se servent **parfois** intelligemment de ces expressions machiavéliques.

Le nom d'une tableau utilisé dans une expression est automatiquement évalué comme étant l'adresse du premier octet de son premier emplacement. Cette adresse est donc celle retournée par « &tab[0] ». Cette caractéristique des tableaux est tout simplement géniale. Le nom d'un tableau, considérée comme une adresse **inamovible**, n'est donc pas une **Lvalue**. Vous ne le verrez donc jamais à gauche d'une assignation donc « tab = … » est une assignation illégale.

Ce qu'on vient de voir avec une paire de crochets, on parle alors de **tableau à une_dimension**, peut se généraliser à **m_dimensions**. Il suffit d'utiliser le nombre voulu de crochets. En fait on dépasse rarement **dimension 2**, les tas de crochets rendant la compréhension du code extraordinairement pénible. Voici par exemple la définition d'un tableau **10 par 12** de réels:

```
double P[8][12] ;
```

« **P[2][5]** » est donc un des réels de ce tableau.

Qu'est ce que P[2] ? Le nom d'un tableau de taille 5! Vous pourrez constater que ce n'est pas un pointeur ordinaire en évaluant « sizeof(P[0]) » et même « sizeof (P) ».

B. Pointeurs

Vous pouvez définir vos propres objets aptes à conserver des adresses, on les appelle pointeurs, qui seront soit des variables, soit des constantes et de ce fait, possèderont aussi, comme tout autre objet, leur propre classe d'allocation. **Notons cependant que le compilateur ne prend pas en charge l'allocation de l'objet pointé c'est-à-dire le fait de réserver un emplacement en mémoire.** Pour déclarer un pointeur, on utilise l'astérisque après le type dans la déclaration d'une variable :

```
type * nom ;
```

où *nom* est un pointeur de type **« type * »** capable de contenir l'adresse d'un objet de type **« type »**. L'initialisation est facultative si le pointeur n'est pas constant.

Mais comment déclare-t-on un pointeur constant ? Vous n'avez qu'à nous suivre. Nous avons d'abord un *riri* de type T : « T riri; ». Grâce à la déclaration « **const T * roro** » nous avons un pointeur *roro* apte à garder l'adresse d'un T constant mais *roro* le pointeur n'est pas constant. Le **const** utilisé en préfixe nous donne un objet pointé constant mais pas un pointeur constant. Et bien mettons le ailleurs… et pourquoi pas en plein milieu de la déclaration comme dans :

T * const roro = &riri;

roro est maintenant un pointeur constant et toute tentative pour modifier *roro* sera refusée.

Les pointeurs comme les objets ordinaires de classe **auto** et **register** ne sont pas initialisés par défaut. Sans valeur correcte d'adresse au départ, tout emploi de l'opérateur d'indirection en lecture ou en écriture conduit aux pires effets. Les pointeurs de classes **extern** ou **static** sont mis à 0 par défaut.

Il existe d'ailleurs une valeur spéciale pouvant être donnée à tout pointeur pour bien indiquer que l'indirection ne peut être utilisée, c'est la valeur **NULL** (ou 0 en C++).

Il existe aussi un type spécial pour les pointeurs, *void **, capable de recevoir l'adresse de tout objet mais sur lequel l'indirection et l'arithmétique sont interdites. Ce type général est utilisé lorsqu'un traitement peut être effectué peu importe le type du pointeur. Ainsi, la fonction **memcpy** de la librairie string.h reçoit trois informations : deux pointeurs void*, les adresses de la source et de la destination dans la mémoire vive, et un nombre d'octets à copier peu importe le type des objets s'y trouvant.

L'usage courant des pointeurs

1. Le parasitage consiste à donner à un pointeur l'adresse d'une variable déjà définie puis changer la valeur de la variable grâce à une indirection du pointeur. Les fonctions du C ne font que des copies des arguments lors d'un appel. Prévoir un pointeur dans la liste de paramètres et lui donner une adresse lors de l'appel de la fonction permet d'obtenir des effets de bord désirés (et désirables!) sur des objets de l'environnement utilisant la fonction. En passant

l'adresse (&V) d'une variable existante V à un paramètre pointeur du type de la fonction V, l'indirection du pointeur permet de modifier V dans une instruction de la fonction.

2. L'allocation programmée, c'est la capacité par le programmeur d'obtenir du système d'exploitation (SE) des blocs de mémoire augmentant d'autant la mémoire déjà allouée au programme en compilation. Le SE possède une réserve d'octets libres dans laquelle le programmeur puise grâce à une demande explicite. Les fonctions *malloc*, *calloc* et *realloc* remplissent cet usage et renvoie des pointeurs génériques *void ** sur le premier octet du bloc offert. Si le pointeur retourné par le SE est NULL, l'allocation a échouée.

L'opérateur *new* du C++ s'applique à un type et nous renvoie donc un pointeur *bien typé* . Si le pointeur retourné par le SE est 0, l'allocation a échouée.

Un bloc d'octets obtenu en allocation programmée n'appartient pas à une classe d'allocation particulière et la remise en liberté de ce bloc d'octets est sous la responsabilité du programmeur. La fonction *free* en C remplie cette tâche.

En C++, on utilise l'opérateur *delete*.

Mais attention, tenter de libérer explicitement une zone de mémoire non précédemment allouée ou déjà « désallouée ", est une erreur et le comportement du SE est alors imprévisible.

Partie III
TYPES DÉFINIS PAR LE PROGRAMMEUR

1. NOTIONS DE BASE

La lecture de types existants n'est pas toujours facile. Il faut cependant savoir que certaines constructions standards sur un type T fournissent des types associés (le symbole « • » marque l'emplacement naturel d'un identificateur):

- Le suffixe * à un type, « T * • », introduit un objet • de type pointeur de T.
- Le suffixe & à un type, « T & • », introduit un objet • de type alias de T (uniquement disponible en C++)
- Les crochets [ ] en suffixe, « T • [ ] », introduisent un objet • de type tableau d'objets T.
- Les parenthèses ( ) en suffixe, « T • (...) », introduisent un objet • de type fonction retournant un objet de type T.
- **const** en préfixe, « const T • », introduit un objet • de type T non modifiable.[5]

On peut bien sûr coupler deux ou plusieurs constructions pour obtenir un nouveau type diablement complexe à déchiffrer. Ainsi « **const** T (* •) (…) », introduit un objet • de type pointeur de fonction retournant un objet de type T non modifiable.

A. Utilisation du typedef

typedef est un mot réservé du C-ANSI permettant au programmeur de donner un nom symbolique à un type déjà existant, il n'introduit pas un nouveau type. En introduisant le **typedef**, la norme ANSI a contribué à faciliter la lecture du code. Ce « petit » ajout amène avec lui plus de fiabilité et contribue à la notion du code « bien structuré ».

Son utilisation est simple : prenez une déclaration correcte, un type suivi d'un nom, et mettez-y **typedef** en préfixe. Vous obtenez la définition d'un nouveau nom pour ce type.

Exemples de base avec typedef

Nous savons que « double * x_ptr ; » définit un pointeur de réel nommé *x_ptr*. Avec **typedef** utilisé comme suit :

```
typedef double * R_ptr ;
```

on obtient un nouveau nom de type, *R_ptr*, pouvant être utilisé pour déclarer un pointeur de réel: « R_ptr y_pt ; » où *y_ptr* est un pointeur de réel et « R_ptr » remplace « double * ». Avec :

[5] Nous avons déjà vu l'utilisation spéciale du const pour les pointeurs où « T * **const** • » introduit un objet • pointeur constant non modifiable.

```
typedef  int grille [50][50] ;
```

on obtient encore un nouveau nom de type, *grille*, qui nous permettra de déclarer très simplement tout tableau de 50 par 50 d'entiers comme dans : « grille ma_grille ; »

B. Utilisation du enum

« **Enum** » permet de définir un nouvel identificateur de type permettant de représenter un sous-ensemble fini d'entiers où chaque élément est nommé par un identificateur unique.

Sans initialisation, un élément prend la valeur incrémentée (+ 1) de son prédécesseur avec 0 par défaut pour le premier élément. Comme pour une constante, la valeur d'un élément est non modifiable.

Exemples avec enum

Voici d'abord la définition d'une première énumération (on y remarque déjà l'application d'une initialisation sur le premier élément) :

```
enum roues{ unicycle=1 , bicycle , tricycle };
```

puis son utilisation dans la définition de la variable « yam » initialisée à la valeur « bicycle »:

```
enum roues yam = bicycle;
```

Voici maintenant la définition des l'identificateurs de type « couleur » et « temperature » grâce à **typedef** :

```
typedef  enum { rouge , vert , bleu } couleur ;
typedef  enum { chaud = 30 , tiede = 15 ,
                froid = 1  , glacial = -20 } temperature ;
```

puis leur utilisation dans la définition de deux variables :

```
couleur plafond  = vert ;
temperature surprise = froid ;
```

Remarquons, pour terminer, que l'assignation « plafond = vert » est tout à fait correcte alors que l'assignation **« vert = 2 »** n'a aucun sens, « vert » étant une **constante !**

Commentaire

L'idée est intéressante mais les moyens mis en œuvre par le langage sont ridiculement simples. Ainsi dans une expression, les variables de ce type sont converties en **int**. De plus, elles ne sont malheureusement pas limitées aux valeurs présentes lors de la déclaration du type énuméré (surprise=567;) et souvent aucune validation des valeurs prises n'est effectuée par le compilateur.

La création de types énumérés reste néanmoins très utile pour représenter informatiquement les valeurs possibles d'une qualité ou d'une caractéristique (la couleur, la marque, le jour de la

semaine, etc.). Ils aident à structurer les programmes avec des identificateurs bien choisis mais d'autres langues impératives les maintiennent beaucoup mieux.

2. LA CRÉATION DE VRAIS TYPES

Si tout était semblable les types resteraient simples. Malheureusement, les problèmes informatiques mettent souvent le programmeur en face de données hétérogènes (de types différents) dont les valeurs décrivent une situation réelle bien précise. Rien de mieux que la diversité.. bien ordonnée. Ainsi, le regroupement des données hétérogènes en **objet** informatique **unique** facilite le traitement et favorise la cohérence en « simulant dans le programme *l'objet réel observé* ».

Vous connaissez déjà des agrégat d'éléments homogènes, les tableaux. Par contre un agrégat d'éléments de types arbitraires est-il envisageable ? Oui bien sûr.

Supposons que nous aimerions construire un programme manipulant « des moyens de transport » (quels qu'ils soient). Ceux-ci devront évidemment mettre en relation plusieurs types de données. Un « véhicule » quelconque possède un milieu d'évolution (air, mer, route, rail....), un fabriquant, un tonnage, une vitesse moyenne, un prix d'achat, un nombre de passagers, un apport énergétique, un numéro de série, une année de fabrication, une espérance de vie, un nombre de membres d'équipage, un coût annuel d'entretien prévu et on en oublie... Le concept de « struct » nous permettra de disposer d'un type dont la variable liera toutes ces caractéristiques pour chaque « véhicule » traité.

A. struct

struct permet la définition d'un nouveau type nommé à l'intérieur duquel on attribuera un nom significatif à chacune de ses parties (on parle de ses champs ou de ses membres).

Il est cependant à remarquer que certains types de champs sont interdits dans la définition d'un **struct chose :**

1. Un struct chose est interdit puisque l'inclusion irait à l'infini!! Mais rien n'empêche de retrouver un champ pointeur de struct chose puisque le compilateur connaît bien la taille d'un pointeur. Cette forme classique permet la création des **listes**. Il va sans dire qu'un struct autreChose est permis puisque dans ce cas la définition du struct chose n'est pas « récursive ».
2. Une fonction est interdite mais pas un pointeur sur une fonction. Historiquement ces membres pointeurs de fonctions font en partie le pont entre le C_ANSI et le C++ objet. Notre cours couvrant C et C++, nous négligerons complètement cette particularité en C pour n'étudier que les classes du C++ (ce sera beaucoup plus propre).

Le nouveau type T possédera les mêmes capacités d'extension qu'un type de base :

- il sera possible de définir des tableaux T ... [ ] de struct
- des pointeurs T * ... de struct
- des constantes const T de struct
- ou des fonctions T ... (...) manipulant ou retournant des structs.

Toutes les associations possibles sont admises.

Syntaxe

```
struct [ nom ]
   { type du membre nom du membre ; ...} [ v1,v2] ;
```

où ...

- [v1,v2] est facultatif et permet d'incorporer la déclaration de variables à la définition du nouveau type *nom*.
- [nom] est facultatif, mais s'il est absent, la nouvelle structure est anonyme et vous perdez la capacité de déclarer des variables ultérieurement. Cette absence n'est permise que lors de la définition, ce qui permet de satisfaire des besoins très limités. À l'inverse, **typedef** est fréquemment employé directement lors de la définition du nouveau type comme dans :

  ```
  typedef struct {mettez-y ce que vous voulez}nom ;
  ```

  et c'est la forme que nous privilégions.
- Chaque membre possède *normalement* un type et un nom.

Les accolades présentes **n'identifient pas** un bloc de code ordinaire. Remarquez le point virgule à la fin ! !

Représentation mémoire

Attention à la représentation en mémoire d'un type structuré, certains types ont des contraintes d'alignement qui force par exemple l'utilisation d'une adresse précise pour l'octet de départ. Ces contraintes amènent la présence d'octets non représentatifs dans le groupe d'octets réservés pour une variable par le compilateur. Dans ce cas, la taille totale du type structuré obtenue par **sizeof** est plus grande ou égale à la somme des **sizeof** effectués sur ses parties.

Cependant, la norme prévoit que

- les adresses des membres respectent l'ordre de leur déclaration dans la structure;
- l'adresse de la variable est la même que celle du premier membre;
- il est possible d'obtenir l'adresse d'un membre avec l'opérateur &.

Dans certaines implémentations du C, lorsque l'adresse 0 admet l'indirection, la macro :

```
offset(type , membre)
```

permet d'obtenir directement le décalage véritable d'octets d'un membre dans la représentation du type structuré. On notera, pour la postérité, que cette macro se déploie ordinairement en :

```
(int) &( (type*) 0 -> membre )
```

Variable et initialisation

On peut initialiser une variable **struct** à la déclaration. Soit :

```
typedef struct  {double Reel ; double Ima ;} complexe ;
```

la déclaration/définition :

```
complexe c1 = { 1.22 , -3.45 } ;
```

est correcte, mais notons cependant que cette forme est interdite en assignation normale.

Accès aux membres

L'opérateur point (.) utilisé en position infixe entre un objet de type structuré et le nom d'un champ permet de lire ou d'écrire (Rvalue ou Lvalue) sur ce champ particulier.

Exemples

1. `complexe c2 ; c2.Ima = -1.7 ;`
   est correct.

2. `complexe * c_ptr = &c2 ;`
   est correcte mais c_ptr *parasite* la variable C2.

Notons pour finir que la simple expression :

```
4 * (* c_ptr).Reel
```

qui doit se lire : « quatre fois la partie réelle du complexe pointé) », et qui utilise l'indirection et l'opérateur point est plutôt rébarbative. Le C permet aussi d'atteindre le champ d'une structure pointée avec l'opérateur « -> ». De sorte que l'expression :

```
(* c_ptr).Reel
```

peut être remplacée agréablement par :

```
c_ptr -> Reel
```

L'expression précédente s'écrira donc :

```
4 * c_ptr -> Reel
```

et devient beaucoup plus facile à lire.

B. union

Les unions sont des structures dont tous les membres sont implémentés à la même adresse. Les unions sont souvent utilisés par souci d'économie d'espace comme membre d'une structure afin

d'éviter de « traîner » des membres « en trop ». C'est souvent le cas lorsqu'on essaie de représenter par un seul type des objets qui sont presque semblables.

Ainsi un véhicule marin a un tirant d'eau, un véhicule routier a un nombre de roues et un véhicule aérien a une envergure, ces trois champs mutuellement exclusifs étant donné le milieu d'évolution gagneraient à être implémentés dans une **union** malgré le fait qu'un opérateur point supplémentaire sera désormais nécessaire pour accéder aux champs. Malheur à celui qui fait varier le tirant d'eau d'un camion ! !

Considérons les déclarations suivantes :

```
typedef  enum { air , eau , terre } milieu ;
typedef struct { milieu m;
                 union {double envergure;
                        int nbRoues;
                        double tirantEau;}prop;
                 } vehicule;


vehicule v1;
```

Si l'on doit accéder au tirant d'eau, au nombre de roues ou à l'envergure, il faut évidemment examiner d'abord à quel milieu le véhicule appartient :

```
if (v1.milieu == eau)
 v1.prop.tirantEau = 2.64;
else if (v1.milieu == terre)
 v1.prop.nbRoues == 10;
```

On se sert aussi marginalement des unions pour jouer directement sur les octets de la représentation d'un type, par exemple :

```
union bits_double { double x ; char tab[8] ; } ;
```

Une variable de ce type offre au programmeur la capacité de jouer directement sur les bits d'un double à travers les éléments du tableau *tab* et des opérateurs de bits ! !

C. bit field (ou champs de bits)

Pour finir, il est également possible d'utiliser des champs de bits implémentés dans un struct. Mais comme ces structures sont souvent inexportables, nous les ignorerons.