Chapitre I

Structuration, typage et localité

Sommaire

1 Introduction

Le C++ a été initialement conçu comme une extension du C.

Comme le C, le C++ est un langage de programmation déclaratif par bloc typé :

Bloc : suite d'instructions exécutée **séquentiellement** (= l'une après l'autre) qui commence par une accolade ouvrante et se termine par une accolade fermante.

Par bloc : la structuration et l'articulation des blocs représentent le code à exécuter.

Typé : Toutes les variables ont un type déterminé ; toutes les fonctions ont leurs entrées/sorties typées.

Déclaratif : Tout objet doit avoir été préalablement déclaré avant d'avoir été utilisé.

Structuration par blocs :

- l'exécution séquentielle peut être modifiée avec :
  - ◇ un **test** (exécution conditionnelle : `if`/`else`, `switch`),
  - ◇ une **structure de répétition** (`for`, `do`/`while`, `while`),
  - ◇ une **rupture de séquence** (arrêt/saut lors de l'exécution d'un bloc : `break`, `continue`, `return`),
  - ◇ un saut inconditionnel (`goto`),
  - ◇ la gestion des exceptions (C++ : branchement en cas d'erreur)
- un **bloc nommé** définit une fonction (exécutée en la nommant).
- un bloc peut contenir d'autres blocs (ou sous-blocs).

Note : en ce sens, identique au C.

Compatibilité C-C++ :
Le C++ essaye de conserver une compatibilité avec le C, lorsque cela est possible, mais il y a des incompatibilités.

En pratique, la plupart des programmes C sont des programmes C++ valides qui peuvent être compilés avec un compilateur C++.

Mais ces incompatibilités continuent à augmenter au fur et à mesure de l'évolution des deux langages,

Exemples d'incompatibilités :
- si le code C utilise des mots-clef du C++ (`new`, `delete`, ...),
- appel d'une fonction sans argument : en C, les arguments ne sont pas spécifiés ; en C++, la fonction n'a pas d'argument ou a des valeurs d'arguments par défaut.
- en C, si le type de retour d'une fonction n'est pas spécifié, c'est `int` ; en C++, il doit toujours être spécifié.
- en C, un pointeur `void*` peut être converti implicitement en tout autre type de pointeur ; pas en C++.
- les structures imbriquées n'ont pas la même portée en C ou en C++

2 Types

Le type est donné au moment de la définition de la variable, constante ou fonction, et avant son utilisation.

Sur un ordinateur, tout symbole stockant une valeur est caractérisée :
- par son **nom** (permet de faire référence à ce symbole dans le code)
- par son **adresse** (= l'endroit où il est stocké dans la mémoire, automatique)
- par son **type** (=comment interpréter les données stockées à cette adresse mémoire).

Le typage a deux buts principaux :
- indiquer comment interpréter la zone mémoire où est stocké le symbole.
- éviter les mélanges invalides de type et permettre les calculs entre types numériques compatibles.

2.1 Modèle de données

Sur un ordinateur, un modèle de données X/Y/Z est caractérisé par la taille respective :
- X des entiers (`int`),
- Y des entiers longs (`long int`),
- Z des pointeurs (par exemple `int*`).

Exemple :

un modèle de données 4/4/8 stocke les entiers sur 4 octets, les entiers longs sur 4 octets, et les pointeurs sur 8 octets.

Il existe essentiellement 4 modèles de données :
- **LP32** = 2/2/4 (système 32 bits) win16 API.
- **ILP32** = 4/4/4 (système 32 bits) win32 API, Linux, Mac OS X.
- **LLP64** = 4/4/8 (système 64 bits) win64 API.

- **LP64** = 4/8/8 (système 64 bits) Linux, Mac OS X.

Les autres modèles sont actuellement très rares.
Attention : la taille des types **entiers standards** et des **pointeurs** doit être considérée **comme variable** car elle peut changer en fonction du processeur **et** du système.

Conséquence : si le modèle de données change, alors :

- la valeur maximale qui peut être stockée dans un entier peut changer.
- la taille de la mémoire nécessaire pour stocker les tableaux ou les structures/classes contenant des types de taille variable peut changer.
- l'alignement dans les structures peut provoquer des trous lors du passage des pointeurs de 32bit à 64bit (voir les propriétés des alignements dans les structures/classes).
- un fichier écrit avec un code utilisant des types de taille variable peut ne pas être lisible sur un autre ordinateur utilisant un modèle de données différent.
- ...

Il convient donc d'être conscient de ces phénomènes et d'utiliser des types à longueur fixe pour toutes les structures dont la taille doit demeurer fixe.

Dans tous les cas, le passage de pointeur 32bit à 64bit (ou vice-versa) peut être la source de problèmes.

2.2 Types élémentaires

Les types dit "élémentaires" du C++ (ou Built-In Type) sont les suivants :

a) Type booléen

`bool` : (8 bits) valeurs `true` ou `false` (mots clef prédéfinis)
Le résultat d'une opération booléenne est un booléen.

Exemple:
```
bool  b0=true, b1 = (4 > 5);
bool  b2 = 6;     // équivalent de 6 != 0  donc vrai
```

NOTES:

- donc un tableau de type `bool[8]` fait 64 bits, et non un octet.
- le conteneur `vector<bool>` est un stockage optimisé pour stocker 1 bit par élément du vecteur, au dépend de la performance.

b) Types caractères

- `char` : type le plus efficace pour représenter les caractères.
- `signed char` : caractère signé.
- `unsigned char` : caractère non signé, représentation mémoire (= type d'un octet).

Les caractères peuvent aussi être utilisés pour stocker et manipuler des entiers sur 8 bits (attention au dépassements).

Syntaxe des littéraux :

- entre 'x' = un caractère, entre "xxx" = une chaine de caractères.
- un caractère peut aussi être défini par sa valeur numérique (décimale, octale, hexadécimale, binaire), voir la syntaxe de littéraux entiers.

Exemple:
```
char  a = 'A';
char  b[5] = "toto";
```

c) Types "modernes" de caractères

- `wchar_t` : caractère long (système unicode :32 bits / windows :16 bits).
- `char16_t`/`char32_t` : type pour représenter l'UTF-16/32.

Syntaxe des littéraux :
u8=UTF-8, u=UTF-16, U=UTF-32, L=wide char.

Exemple : `wchar_t s=L't';`

d) Types entiers à longueur variable

`int` : type entier de base (32 bits, signé)

modificateurs :
- `signed`/`unsigned` = signé/non signé
- `short`/`long`/`long long` = au moins 16/32/64 bits.

Exemples de types modifiés :
- `short int` : 16 bits
- `int` : 32 bits, sauf LP32=16bits,
- `long int` : 32 bits, sauf LP64=64bits,
- `long long int` : 64 bits (C_{11}^{++})

Syntaxe littéraux :
postfixe : rien=int, u/U=unsigned, l/L=long/long long, ll/LL=long long.
préfixe : rien=décimal, 0=octal, 0b/0B=binaire, 0x/0X=hexadécimal.

Exemple:
```
long long int  a=1000000000000LL;
int            b=0xf245;
```

e) Types entiers à longueur fixe (C_{11}^{++})

- `int8_t`, `int16_t`, `int32_t`, `int64_t` : type entier signé de 8, 16, 32, 64 bits exactement.
- **variation :** `int_`x`8_t`, `int_`x`16_t`, `int_`x`32_t`, `int_`x`64_t` avec x=
  `fast` : type entier le plus rapide avec ce nombre de bits.
  `least` : plus petit type entier avec au moins ce nombre de bits.
- `intmax_t` : type entier le plus long.
- `intptr_t` : type entier assez long pour stocker un pointeur.
- tous les types précédents précédés de `u` = version non signée (exemple : `uint32_t` = entier non signé sur 32 bits).

Note : défini dans <cstdint>.

Exemple:
```
uint8_t       a=4;   // entier non signé sur 8 bits.
int_fast16_t  b=430; // entier rapide 16 bits minimum.
```

f) Types flottants

- `float` : flottant simple précision (32bits, IEEE-754), LSB/MSB $\simeq 10^{-7}$
- `double` : flottant double précision (64bits, IEEE-754), LSB/MSB $\simeq 2.10^{-16}$
- `long double` : flottant précision étendue (80bits), LSB/MSB $\simeq 2.10^{-19}$.
  non implémenté par VCC14 (i.e. `long double` = `double`).

Note : LSB/MSB est le rapport entre le LSB et le MSB pour ce type. Pour un `float` cela signifie par exemple que $1.0 + 10^{-8} = 1.0$ exactement.

Syntaxe littéraux :

rien=double, f/F=float, l/L=long double

Exemple:

```
float       a=1.3f;
double      b=1.5;
long double c=1.21L;
```

Note : Tout non respect de la syntaxe des littéraux pour un type entraine des conversions numériques.

2.3 Propriétés des types numériques élémentaires

En fonction du modèle de données, les propriétés des types élémentaires peuvent changer en fonction de la plateforme utilisée.

Conséquence : nécessité de disposer d'un moyen de connaître les caractéristiques d'un type élémentaire :

- pour les entiers : min/max
- pour les flottants : min/max, epsilon, erreur d'arrondi maximale, codage de inf/NaN, ...

Deux méthodes en C++ :

- par des constantes de <climits> (entier,char) et <cfloat> (flottant)
  Exemple : `INT_MIN`, `INT_MAX`, `FLT_EPSILON`, ...
  Idem en C dans <limits.h> et <float.h>.
- par des fonctions génériques (C_{11}^{++}) dans <limits> à travers le namespace `std::numeric_limits<T>`.
  Exemple : `std::numeric_limits<int>::max()`.
  Donne un moyen d'accès à la constante sans connaître son nom.
  Note : `constexpr` depuis C_{11}^{++}.

Méthodes de `numeric_limits<T>` :

- **Principales méthodes utiles pour les entiers/caractères**
  `min()`/`max()` plus petite/grande valeur.

- **Principales méthodes utiles pour les flottants**

`min()`/`max()`	plus petite/grande valeur.
`epsilon()`	LSB/MSB.
`round_error()`	erreur d'arrondi maximale.
`infinity()`	codage de +inf dans le type.
`quiet_NaN()`	valeur de NaN (sans exception).

2.4 Types énumérés

Deux types d'énumération :

- énumération classique (unscoped enum), dans lequel chaque mot de l'énumération défini un symbole global.
- énumération délimitée (scoped enum) dans lequel chaque mot de l'énumération est défini dans l'espace local associé à l'énumération.

Exemple:

```
// unscoped enum
enum Color { black, white, red };
auto white = false; // error: white déjà dans la portée
// scoped enum
enum class Color { black, white, red };
auto white = false; // ok, white symbole local bool
Color c = white; // erreur, white pas dans cette portée
Color c = Color::white;  // ok
auto c = Color::white;  // ok
```

Autre problème : exemple d'une énumération classique est implicitement convertie en entier, et depuis là, en flottant (pas de warning).

Exemple:

```
// Color unscoped enum
Color c = red;
if (c < 7.3) { ... } // comparaison couleur et double!
auto fact = Factorial(c); // factorielle d'une couleur!
```

Le type par défaut d'un `enum` est `int`.

Les énumérations délimités peuvent aussi être typée :

```
enum class EnumName:Type  EnumList ;
```

Exemple:

```
// color red (=0), yellow (=1), green (=20), blue (=21)
// defaut = int
enum color { red, yellow, green = 20, blue };

// enum 16 bits (a=0, b=1, c=2)
enum smallenum: int16_t { a, b, c };

// altitude altitude::high ou altitude::low
enum class altitude: char {
     high='h',
     low='l',
};
```

Conclusion : il est préférable d'utiliser une énumération délimité.

2.5 Types agrégés

En C++, un type agrégé est :

- un type personnalisé permettant d'agréger dans un même conteneur plusieurs variables différentes de types différents (= champs).
  extension de `struct` en C,
  une instance de ce type s'appelle un objet.
- d'inclure des fonctions au conteneur (appelée méthode) ne s'appliquant qu'à un objet de ce type,
- d'avoir des méthodes qui s'exécutent automatiquement à la création/copie/destruction d'objet,
- d'ajouter des restrictions d'accès aux champs et aux méthodes,
- par le mécanisme d'héritage, de créer des structures de données plus complexe.

Deux façons de créer des types agrégés :

- `struct` : comme en C, avec méthodes en plus,
- `class` : struct avec restriction d'accès

Les types agrégés seront abordés en détail ultérieurement.

2.6 Modificateurs de type

Un type T peut être modifié avec les modificateurs suivants :

- `T*` : pointeur sur un `T` (=adresse mémoire dans laquelle est stockée une variable de type `T`).
- `T&` : référence sur une lvalue de type `T` (=alias d'un objet nommé).
- `T&&` : référence sur une rvalue de type `T` (=alias d'un objet temporaire).

Exemple:

```
int  a;          // entier classique
int  *b = &a;    // pointeur sur un entier
int  &c = a;     // référence à l'entier a
int  &&d = 4+a; // référence au résultat de 4+a
```

Le sens précis de ces modificateurs sera spécifié au fur et à mesure.

Attention :

- Un **type modifié** est un **type différent** potentiellement incompatible sauf si un sens peut être donnée à sa conversion dans le type d'origine.
- Un objet de type `T&` ou `T&&` peut toujours être utilisé dans une LHS à la place d'une variable de type `T`.
- Un `T&` ou un `T&&` est en interne similaire à un pointeur `T*`.
- `sizeof(T*)` = définie par le modèle de données (= identique pour tous les types, donc indépendant de `T`).

2.7 Qualificateurs de type

Un qualificateur de type est un spécificateur qui permet de modifier le traitement du type qualifié par le compilateur :

- `const` : indique que le type est constant (= non mutable), à savoir toute tentative de modifier un objet qualifié `const` provoque une erreur de compilation (reprécisé plus tard).

- `volatile` : indique que pour l'objet dont le type est marqué volatile, le compilateur ne peut pas réordonner ses lectures/écritures lors de l'optimisation (typiquement pour des variables partagées entre threads).
- `mutable` : dans un objet `const`, indique que ce membre est modifiable.

Une qualification de type ne modifie donc pas le stockage mais le traitement des objets qualifiés par le compilateur.

Exemple:
```
const int  a = 3;    // constant
// a = 4;            // erreur
volatile int b = 3;
// lecture/écriture dans b non déplaçable
// i.e. la lecture de b ne se fait pas plus tôt
// l'écriture de b ne se fait pas plus tard
b += 4;
```

Par défaut, pour un objet `T *t`, si `t` pointe sur une adresse mémoire valide :

- dans une LHS, `*t` écrit à l'adresse mémoire pointée par `t`.
- dans une RHS, `*t` lit la valeur stockée à l'adresse mémoire pointée par `t`.

Exemple : pour un objet `x` de type `T`, écriture : `*t = x`, lecture : `x = *t`.

`const` permet de modifier le comportement d'un paramètre. Pour un type pointeur `T*` :

- `T*` : pointeur et valeurs pointées modifiables
- `const T*` : pointeur modifiable, valeurs pointées constantes
- `T* const` : pointeur constant, valeurs pointées modifiables.
- `const T* const` : pointeur constant et valeurs pointées constantes

Attention : c'est la position de * par rapport aux `const` qui détermine le comportement. A savoir, `const int*` = `int const*`.

Pour résumer :

Type	lire v	modifier v	lire *v	modifier *v
`int *v`	oui	oui	oui	oui
`const int* v`	oui	oui	oui	non
`int* const v`	oui	non	oui	oui
`const int* const v`	oui	non	oui	non
`const int& v`	oui	non	n/a	n/a

2.8 Nouveaux types

Applications :

- avoir un nom de type plus court à écrire,
- type composé ou complexe (structure, détaillé plus tard)

Définition des nouveaux types :

```
typedef [définition] [nom]
using [nom] = [définition]     (C++11)
```

Exemple :

```
// déclaration C/C++
typedef unsigned int  uint;
typedef unsigned char byte;
```

```
// déclaration C++11
using uint = unsigned int;
using byte = unsigned char;
```

```
// utilisation
uint  a=3;
byte  b=7;
```

Notes :

- pour le compilateur, `typedef`/`using` ne génère pas de nouveau type. Il se contente de donner un nom simplifié. Les types sont comparés sur la base de leurs définitions exprimées à partir des types élémentaires et des modificateurs.
- `using` permet aussi de déclarer des types génériques (voir la leçon sur les patrons de classe).

3 Conversions

conversion = opération qui transforme un donnéee de type T_1 en une donnée de type T_2.

Il est important de comprendre :

- quelles écritures engendrent des conversions ?
- quels contextes engendrent des conversions ?
- quelles sont les conversions autorisées ?
- comment ces conversions sont-elles effectuées ?
- comment forcer une conversion ?

Sinon, votre code :

- passera du temps à effectuer des conversions inutiles,
- risque de faire autre chose que ce que vous pensez qu'il fait,
- risque de manipuler des données incorrectes.

3.1 Conversions implicites

Une conversion implicite à lieu lorsqu'une expression de type $\texttt{T}_1$ est utilisée dans un contexte où un type $\texttt{T}_2$ est attendu :

Exemples :

- lors de l'appel d'une fonction $\texttt{f(T}_2\texttt{)}$ avec un argument de type $\texttt{f(T}_1\texttt{)}$,
- dans une expression : `a+b` où `a` et `b` sont de types respectifs $\texttt{T}_1$ et $\texttt{T}_2$.
- lors de l'initialisation d'un type $\texttt{T}_2$ avec un type $\texttt{T}_1$ (exemple : $\texttt{T}_2$ `b = a`), y compris dans le retour d'une fonction (= retour d'une type $\texttt{T}_1$ alors qu'un type $\texttt{T}_2$ est attendu ;
- quand un type intégral $\texttt{T}_2$ est attendu dans un `switch`, avec une expression de type $\texttt{T}_1$ utilisée,
- quand une expression de type $\texttt{T}_1$ est utilisée dans un test ou dans une boucle (*i.e.* $\texttt{T}_2$ = `bool`).

Le compilateur tente alors de convertir implicitement le type $\texttt{T}_1$ en type $\texttt{T}_2$ puisque cette conversion résulte d'une instruction explicite du programmeur.

a) Promotion numérique

Dans cette partie, nous n'abordons que les promotions numériques ou les conversions numériques, d'autres types de conversion peuvent également être à l'œuvre (catégorie, pointeur, qualification).

Promotion numérique : conversion qui consiste en la conversion d'un type en un autre type permettant de stocker le type original *sans perte de précision* (=SPP).

- signed char/short → `int`
- unsigned char/short → `int` si SPP, `unsigned int` sinon.
- `int` → `long` → `long long int`
- `unsigned int` → `unsigned long` → `unsigned long long`
- `bool` → `int` (`false`=0, `true`=1).
- également pour les énumérations.
- `float` → `double`.

Toutes ces promotions numériques, si elles découlent du contexte résultant des écritures, sont considérées comme naturelles, et n'engendrent jamais aucun avertissement.

Pour tout calcul mené sur des types entiers plus petits qu'un `int` est implicitement converti en `int` pour les calculs (par promotion numérique), puis reconverti dans le type du résultat.

A savoir, pour les types suivants (ILP32/LLP66/LP64) :

- `char`, `unsigned char`, `short int`, `short unsigned int`,
- `int_8t`, `int_16t`, `uint_8t`, `uint_16t`, et toutes les variations sur 8 et 16 bits

tout calcul numérique (y compris les opérations binaires) est mené en `int` (parfois en `unsigned int`), après conversion des opérandes en `int`.

Exemple:
```
signed char cresult, c1 = 100, c2 = 3, c3 = 4;
cresult = c1 * c2 / c3;
```

L'exemple ci-dessus produit 3 conversions numériques en `int`, puis le résultat est reconverti en `signed char`.

Conséquence : les types entiers à longueur fixe ont donc pour but de rendre certaine la place de stockage utilisée par une variable dans la mémoire, **et non** de permettre des calculs plus rapides. Il faut utiliser des instructions spécifiques MMX/SSE/AVX de calcul vectoriel dans ce but (non encore dans le standard).

b) Conversion numérique

Une **conversion numérique** consiste en la conversion d'un type en un autre type *avec une perte potentielle de précision* [1].

conversion entière :

- vers un entier non signé avec de n bits = source modulo 2^n,
- vers un entier signé : même valeur si peu être représenté dans ce type, sinon indéfini.
- vers un flottant : valeur flottante la plus proche de l'entier, sinon résultat indéfini.

conversion flottante :

1. Nous ne sommes donc pas dans le cadre d'une promotion numérique

- vers un autre type flottant : si le nombre peut être représenté ⇒ perte de précision, sinon résultat indéfini.
- vers un entier : partie fractionnaire perdue (*i.e.* troncature vers 0)

conversion booléenne :

- depuis un booléen, `false`=0 et `true`=1 converti dans le type de destination.
- vers un booléen, réprésentation de 0 = `false`, toute autre valeur = `true`.

c) Conversions arithmétiques

Règles :

- les calculs sont toujours menés dans le type permettant la plus grande précision possible (avec conversion automatique des valeurs si besoin).
  - ◇ calculs avec entiers et flottants effectués en flottant
  - ◇ calculs avec flottants simple et double précision effectués en double précision.
- si un type numérique est affecté à un autre, une conversion automatique a lieu.
  Ceci peut avoir lieu souvent (affectations =, passage de paramètres, retour de paramètre)

Conséquences :

- Évitez d'effectuer des opérations numériques entre des valeurs numériques de types différents.
- Choisir le type **avec soin** et **de manière consistante**.
- Aucune conversion ne doit être implicite (*i.e.* elles doivent toutes être explicites).
- Ne pas croire que le compilateur comprend ce que vous voulez écrire.
  Exemple : `float eps=1/100;` (effectue une division entière, et convertit le résultat en flottant donc `eps=0.f`).

Sinon vous risquez de générer un très grand nombre de conversions numériques implicites, et d'effectuer d'autres calculs que ceux que vous pensez effectuer.

3.2 Conversions et compilation

Notes : les conversions nécessaires à l'exécution sont **déterminées à la compilation**,

- les promotions ne sont pas signalées (car se font sans perte de précision),
- les conversions conduisant à une possible perte de précision engendrent des warnings à la compilation (`int` ⇔ `float`, `int` ⇔ `int`).
- pour les conversions numériques entre entiers (hors promotion), un warning n'est pas toujours affiché.

Exemple : lors d'une opération du type `int` = `float * int`, les calculs sont menés en flottant, puis reconverti en entier : cette écriture génère donc 2 conversions.

Warnings des compilateurs associés aux conversions :

- jamais lorsque cette conversion s'effectue au bénéfice de la précision.
- dans tous les autres cas :
  - ◇ warning affiché par défaut sous Visual Studio.
  - ◇ pas de warning avec `g++` sauf à les forcer avec l'option `-Wconversion`.

3.3 Conversions explicites

Règles sur les conversions :

- Toujours s'assurer que le compilateur signale les conversions avec perte de précision, en ajoutant si besoin les options nécessaires à la compilation.
- Résoudre ces problèmes,
  - ◇ soit en utilisant un type approprié permettant d'éviter la conversion,
  - ◇ soit en spécifiant que cette conversion est souhaitée par l'utilisation d'une **conversion explicite**.

  Dans tous les cas, aucun warning signalant des problèmes de conversions ne doit subsister.

Comment rendre une conversion numérique explicite ?

a) Conversions explicites en C

En C, une conversion numérique explicite s'effectue avec un cast à savoir `(T)x` signifiant que `x` est converti en type `T`.

Exemple : `int a = (int)logf(1000.f);`

Inconvénient : peut convertir presque tout en n'importe quoi.

L'exemple qui suit est un code C valide.

Exemple:

```
int   a = (int)"abcdef";   // sens?
float b = *(float*)&a;     // réinterprétation
```

Un cast permet donc des conversions :

- entre sémantiques incompatibles,
- sans engendrer le moindre message d'avertissement de la part du compilateur, puisque la conversion est explicite (=exigée par le programmeur).

Conséquences :

Le cast du C est donc une fonctionnalité **potentiellement dangereuse** et capable de faire bien plus que ce que l'on lui demande.

b) Conversions explicites en C++

Le C++ introduit un ensemble de méthodes de conversions de type.

La solution : utiliser `static_cast` pour lequel il est garanti qu'il n'effectue **que** des conversions numériques vers des types compatibles (si les types sont incompatibles, la compilation s'arrête sur une erreur).

syntaxe : `static_cast<U>(x)` où `x` est l'expression à convertir et `U` le type dans lequel il doit être converti.

Notes :

Dans le cas des littéraux, il peut être acceptable d'utiliser la forme `T(x)` (équivalent C++ de `(T)x`) **lorsque les types sont fixes**.

A savoir, ne **jamais** utiliser ce type de cast si le type d'origine ou de but est un type variable, *i.e.* issu d'un patron de classe ; ce qui peut arriver dès que le type d'une variable explicitement convertie est changé.

Exemple:
```
// conversion numérique sûre
int   a = static_cast<int>(logf(1000.f));
// conversion numérique non sûre
// (connue comme sûre dans ce contexte)
// mais inutile dans ce contexte
float b = logf( float(1000) );
```

4 Classes de stockage

Tout objet nommé en C++ (variable, constante, fonction, ...) a deux caractéristiques :
- sa **durée de stockage** qui représente le temps de vie de l'objet lors de l'exécution.
- son **lien** qui indique où l'objet peut être trouvé dans le contexte courant.

Elles constituent la **classe de stockage** de l'objet.

Les classes de stockage sont importantes pour comprendre des notions comme la localité, la portée des variables, ou la résolution de noms.

4.1 Unité de traduction

Une unité de traduction (translation unit) est l'entrée reçue par un compilateur issue d'un code source (`.cpp`) dans lequel l'ensemble des commandes du préprocesseur ont été appliquées (inclusion des `.h` compris).

Une fois ceci réalisé :
- si le projet n'a qu'un seul code source, il n'y a qu'une seule unité de traduction.
- si le projet est constitué d'un ensemble de codes sources, il y a autant d'unités de traduction que de sources (`.cpp`).

Dans une unité de traduction, il convient de faire clairement la différence entre :
- la **définition** d'un objet : l'objet est complètement défini dans l'unité de traduction.
  Exemple : pour une fonction, elle est définie avec son code.
- la **déclaration** d'un objet : seule la déclaration de l'objet est faite dans l'unité de traduction, sa définition étant effectuée ailleurs dans une autre unité.
  Exemple : pour une fonction, seul son prototype est donné.
- l'**utilisation** d'un objet : où l'on utilise un objet dans l'unité de traduction qui a été préalablement défini ou déclaré (rappel : obligatoire en C++).
  Exemple : pour une fonction, l'appel de la fonction.

Des exemples concrets seront donnés dans la section sur la structuration d'un code.

4.2 Durée de stockage

Il y a 4 durées de stockage possibles pour un objet :
- **automatique** : si la durée de vie de l'objet est limitée à la section de code dans laquelle l'objet est défini.

- **statique** : si l'objet est alloué lorsque le programme commence et est désalloué lorsque le programme se termine.
- **par thread** : si l'objet est alloué lorsque le thread commence et est désalloué lorsque le thread se termine.
- **dynamique** : si l'objet (i.e. la mémoire qui lui est affectée) est alloué ou désalloué en utilisant les fonctions d'allocation dynamique.

Tout objet dans un code a nécessairement l'une de ces durées de stockage. Il n'en existe pas d'autre.

4.3 Liens

Il y a 3 types de lien possibles pour un objet :

- **pas de lien** : si l'objet est défini dans la portée dans lequel il est utilisé.
- **lien interne** : si l'objet est défini dans l'unité de traduction dans laquelle il est utilisé.
- **lien externe** : si l'objet est défini dans une autre unité de traduction que celle dans laquelle il est utilisé.

Tout objet dans un code a nécessairement l'un ces liens. Il n'en existe pas d'autre.

Note : une bibliothèque compilée (`.a`,`.lib`,`.dll`,...) qui contient la définition compilée d'objets est également, à l'origine, issue d'une unité de traduction. Son fichier d'entête associé (`.h`) contient la déclaration des objets internes à la bibliothèque, et permet leur utilisation externe dans une autre unité de traduction.

4.4 Modificateurs

Les modificateurs suivants permettent de modifier la classe de stockage d'un objet :

- `static` : rend la durée de stockage de l'objet statique (lien interne par défaut),
- `extern` : spécifie que le lien est externe, à savoir qu'il est défini dans une autre unité de traduction (possible sur les variables ou les fonctions).
  Noter qu'un objet défini comme externe dans une unité de traduction ne peut contenir que sa déclaration = prototype de la fonction (donc sans son code) ou variable sans initialisation.
- `thread_local` (C_{11}^{++}) : durée de stockage locale au thread.

Attention :

- avant le C_{11}^{++}, le mot-clé `auto` indiquait qu'un objet avait une durée de stockage automatique (mais inutile car ceci est généralement déterminé par le contexte).
  Attention : il prend maintenant un sens très différent que nous aborderons plus tard.
- le mot-clé `register` sera obsolète dans le C_{17}^{++}.
  Il servait à suggérer au compilateur de placer la valeur de l'objet dans un registre du processeur. L'évolution des optimisations dans les compilateurs rends désormais ce spécificateur inutile.

5 Variables

Une **variable** est un objet qui a :

- un nom unique (= nom du symbole)
- un type (= comment interpréter la zone de mémoire qui stocke le symbole) avec ses modificateurs et qualificateurs.

- une portée (= portion du code dans laquelle le symbole est défini)
- éventuellement, une initialisation (= valeur de la variable à l'initialisation)
- une classe de stockage

Une **constante** est une variable dont la valeur ne peut pas changer (*i.e.* elle doit être définie à la déclaration).

Approfondissons d'abord la notion de portée.

5.1 Localité et portée

a) Variable globale

Une variable définie hors de tout bloc a les propriétés suivantes :

- **classe de stockage :** lien interne, statique
  lien externe possible dans une autre unité de traduction.
- son nom doit être unique dans toutes les unités de traduction (i.e. définie dans l'une des unités de traduction, lien externe dans les autres).
- elle est utilisable dans tout bloc (donc, dans toute fonction).
- elle est la seule variable globale qui porte ce nom dans toute l'application.

Ce type de variable s'appelle une **variable globale**.

Important :

- (redite) une variable globale est allouée lorsque le programme commence et desallouée lorsqu'il se termine.
- une variable globale est initialisée par défaut à 0 si une initialisation n'a pas été fournie.
- l'usage de variable globale est strictement déconseillé,
- en C++, elles peuvent **toujours** être évitées (voir singleton).

b) Variable locale

Une variable définie dans un bloc a les propriétés suivantes :

- classe de stockage : automatique, pas de lien
  = elle n'est définie que dans ce bloc
- elle n'est utilisable qu'à partir du moment où elle est définie.
  en C++, n'importe où dans un bloc (= avant utilisation).
  règle : permet toujours d'initialiser la variable à sa création.
  sinon sa valeur n'est pas définie (*i.e.* valeur inconnue).
- elle est utilisable dans tout sous-bloc du bloc qui contient et suit sa définition.
- elle est définie jusqu'à la fin du bloc (désallouée après).
  note : après la fin du bloc, impossible de l'utiliser, interdit de faire référence à la zone mémoire qui contenait la variable.
- elle est la seule variable du bloc qui porte ce nom.

Cette variable s'appelle une **variable locale**.

Exemple :

```
// unitA (fichier unitA.cpp)
//////////////////////////

// var. globale
int a=5;

// fonction externe (unitB)
extern void fun();

int main() {
   // var. locale
   int   b=3;
   ...
   // idem en milieu bloc
   int   c; // valeur inconnue
   fun();
   ...
}  // fin portée: b,c
```

```
// unitB (fichier unitB.cpp)
//////////////////////////

// var. global: valeur=0
int u;
// lien externe (unitA)
extern int a;

void fun() {
   // var. locale
   int v=5;
   ...
   {
      // var. locale
      int w=14;
      ...
   } // fin portée w
} // fin portée v
```

c) Variable statique

Une variable statique est une variable locale définie avec le mot-clef `static`.

Elle a les mêmes propriétés comme une variable locale, sauf qu'elle est allouée comme une variable globale à savoir :

- elle est allouée lorsque le programme commence,
- elle est initialisée une seule fois au moment de son allocation,
- si aucune valeur n'est donnée pour son allocation, elle est initialisée à 0.
- il est valide de renvoyer l'adresse d'une variable statique.

Conséquence : une variable locale statique conserve sa valeur après la fin de l'exécution du bloc dans lequel elle est définie.

Exemple :

```
int count() {
   static int i;
   return ++i;
}
```

```
int main() {
   int a=count();
// a=1
   a=count();
// a=2
}
```

Notes :

- le mot-clef statique sur une variable globale n'a aucun effet, puisqu'elle est par essence statique.
- il est possible de retourner l'adresse d'une variable locale statique à l'extérieur du bloc qui la définie.

d) Localité et portée des variables

On appelle :

- la **portée d'une variable** : l'ensemble des blocs sur lesquels elle est définie.
- la **localité d'une variable** : l'ensemble des blocs sur lesquels elle est utilisable.

Pourquoi cette différence ?

Il existe des cas où la portée et la localité d'une variable ne se recouvrent pas.

Il est possible de définir deux variables portant le même nom dans des blocs différents, mais telles que les portées de ces variables ne soient pas disjointes.

5.2 Occultation

Dans ce cas, il y a **occultation** (shadowing) des variables.

La règle est alors la suivante : le symbole fait toujours référence à la variable la plus locale.

Exemples :

- une variable locale occulte une variable globale de même nom.
- une variable locale d'un sous-bloc occulte un variable locale de même nom défini dans tout bloc parent.
- une variable locale d'une fonction occulte un paramètre de même nom (provoque en général une erreur/un warning du compilateur)

Les mêmes noms donnés aux variables peuvent mener à une confusion sur la variable qui est véritablement utilisée.

Attention :

- l'occultation d'une variable ne produit **ni erreur**, **ni warning** sauf dans le cas de l'occultation d'un paramètre, même en mode `-Wall`.
- c'est un phénomène **normal** car une variable locale est faite pour être utilisée **localement**, à savoir, à proximité de l'endroit où elle est définie.

Exemple:

```
int a=4;  // [1] var. globale          // a#1
                                       //  |
void fun() {                           //  |
   a++; // incr. [1]                   //  |
   {                                   //  |
      a++; // incr. [1]                //  |
      ...                              // a#1
      int a=10;  // [2] var. locale    // a#2
      {                                // a#2
         int a=7; // [3] var. locale   // a#3
         ...                           //  |
         a++; // incr. [3]             // a#3
      }  // déallocation [3]          // a#2
      ...                              //  |
      a++; // incr. [2]                // a#2
   } // déallocation [2]               // a#1
   ...                                 //  |
}                                      //  |
                                       //  |
void fun2() {                          // a#1
   int a=4; // [4] var. locale         // a#4
   ...                                 // a#4
} // déallocation [4]                  // a#1
```

5.3 Localisation

Il est très vivement conseillé d'adopter les règles suivantes pour les variables locales :

- une variable locale = un usage (on donnera un nom approprié ; les EDIs modernes savent faire de la complétion automatique pour les noms longs).
- déclarer la variable locale au plus près de l'endroit où elle doit être utilisée, si possible en l'initialisation à sa creation.
- localiser la variable, au besoin en créant un sous-bloc permettant de limiter sa portée.
- un compteur de boucle doit être une variable locale, localisée à la boucle et utilisée seulement pour cette boucle.
  en C++, déclaration la variable de boucle dans le `for`.

Exemple : pour les boucles

```
// i locale au for
for(int i=0;i<10;i++) {
    ...
}
// v rendue locale au do/while
{ bool v;
    do { ...
    } while (v == false);
}
```

```
{ // bloc dans la  fonction courante
    ...
    { // calcul spécifique
        int somme=0; // local pour ce calcul
        ...
    } // somme n'est plus nécessaire ici
    ...
}
```

6 Constantes

Il existe trois manières possibles de définir une constante en C++ :

- avec un symbole du préprocesseur (C)
- en déclarant une variable avec le qualificateur `const` (C++),
- en déclarant une expression constante `constexpr` (C^{++}_{11})

Il est important de bien comprendre les nuances entre ces différentes déclarations car :

- chacune à sa manière représente la sémantique d'une constante,
- leurs traitements par le compilateur n'est pas du tout le même.
  induit des nuances sur ce en quoi la constante est constante.

6.1 Constantes du préprocesseur

Rappel : en C, `#define NOM VAL` définit un symbole du préprocesseur `NOM` dont la valeur est `VAL`.

Exemple : `#define PIf 3.1416f`

Cette déclaration utilise le préprocesseur (= à partir de la définition, rechercher `NOM` et remplacer toutes les instance de la chaine de caractères `NOM` par `VAL`, jusqu'à la fin de l'unité de traduction, où d'un `#undef NOM`).

Conséquences :

- aucun autre symbole ne peut avoir le même nom y compris localement, par exemple dans une fonction.
- typiquement cette déclaration est **globale** à l'unité de traduction mais non exportable,
- assure l'évaluation et l'optimisation par le compilateur.

- problème : cette déclaration est **non typée**.

6.2 Variables non mutables comme constantes

Rappel : le qualificateur `const` permet d'indiquer que la valeur d'une variable ne peut être modifiée après sa création (= variable non mutable).

Exemple : `const float PIf = 3.1416f;`

Conséquences :

- la valeur est typée et exportable (si elle n'est pas temporaire).
- la valeur est non mutable, mais peut être le résultat d'une évaluation à l'exécution (i.e. peut être initialisée à la compilation ou à l'exécution),
- le compilateur peut optimiser une constante affectée avec une valeur numérique constante (i.e. remplacer par la valeur constante dans le code).
- dans les autres cas, traitée comme une variable non modifiable.
  attention l'optimisation peut faire occuper à différentes constantes le même espace mémoire (cf partie texte d'un code).

6.3 Expressions constantes

a) Qualificateur `constexpr`

`constexpr` est un qualificateur permettant d'indiquer qu'une **variable** ou une **fonction** peut apparaître dans une expression constante qui peut être évaluée à la compilation.

Exemple:

```
constexpr int n=3;
constexpr int two(int p) { return 2*p; }
constexpr int n2 = two(n);  // évaluée à la compilation
```

Conséquences :

- permet de forcer des déclarations de constantes dont on oblige l'**évaluation à la compilation**
- `constexpr` sur une variable implique `const`.
- le type d'une variable `constexpr` doit être littéral
  = possibilité de le construire immédiatement ET lui affecter une valeur,
  + les paramètres du constructeur doivent être uniquement des valeurs littérales, des variables ou fonctions `constexpr`.
- une fonction `constexpr` est une fonction dont les résultats peut être évalué à la compilation et qualifiée `constexpr`.

Note : littéral = scalaire, type tel que destructeur trivial + type agrégé ou constructeur constexpr + membres non volatiles (sauf statiques).
Ainsi avec le C++, une constante qualifiée `const` ou `constexpr` a une portée :

- locale si elle est déclarée dans un bloc,
- globale si déclarée à l'extérieur de tout bloc.

La valeur d'une expression constante à la compilation peut être utilisée par le compilateur seulement si elle est initialisée par un littéral (possible avec `const`, garantie par `constexpr`).

Exemple :

```
constexpr int two(int p) { return 2*p; }
const int a=3;
constexpr int b=2;
const int c = two(a);       // éval. a priori à l'exécution
constexpr int d = two(b);   // éval. à la compil. obligatoire
// dans ce dernier cas: dépend de la capacité de two à
// produire une expression constante à partir d'une
// expression constante
int t1[a];          // ok car a constant init. par littéral
int t2[b], t4[d]; // ok car b et d constexpr
// int t3[c];       // erreur: c évaluée à l'exécution
```

b) Variables `constexpr`

Exemple :

```
constexpr int x = 42;
int y = 1;
x = 0;                // ERROR: x is const
const int& x1 = x;    // OK
const int* p1 = &x;   // OK
int& x2 = x;          // ERROR: x const, x2 not const
int* p2 = &x;         // ERROR: x const, *p2 not const
int a1[x];            // OK: x is constexpr
int a2[y];            // ERROR: y is not constexpr
```

Le code ci-dessus produit le même résultat en remplaçant `constexpr` par `const` car `x` est initialisé avec un scalaire.

Résumé

- une variable `constexpr` est aussi `const` (à savoir, si on utilise son adresse ou une référence à sa valeur, elle se comporte comme un `const`).
- une variable `constexpr` peut être utilisée partout où une constante littérale est attendue (à taille d'un tableau statique, paramètre entier d'un template, ...).

7 Fonctions

Rappel :

- une fonction est un bloc nommé que l'on peut exécuter avec son nom, dont les paramètres permettent de passer des valeurs au bloc lors de son appel, dont le type de retour de la fonction permet de renvoyer une valeur (ou non si ce type est `void`),
- si la fonction renvoie une valeur, tous ses chemins d'exécution possibles aussi.

Le C++ apporte les innovations suivantes :

- la surcharge de fonctions (possibilité d'avoir le même nom pour plusieurs fonctions différentes),
- les valeurs par défaut aux paramètres,
- la qualification des paramètres,
- de nouveaux modes de passage de paramètre avec les références.

7.1 Nommage et surcharge des fonctions

Rappel : en C, seul le nom est utilisé pour différencier les fonctions.

En C++,

- deux fonctions sont différentes si leurs noms sont différents ET leurs paramètres sont différents. Dans ce cas, on dit que la fonction est **surchargée** (= plusieurs fonctions avec le même nom existent avec des arguments différents).
  Attention : le type de retour n'est pas pris en compte.
- un type surchargé ne peut pas être seulement différent en constance (*i.e.* `fun(T)` et `fun(const T)`), les règles sont précisées ci-après.
- lors de la compilation, c'est le type des arguments qui permet de déterminer quelle fonction doit être appelée.

Exemple 1 :

```
// deux fonctions différentes par le type
unsigned int div2(unsigned int x) { return x >> 1;  }
float div2(float x) { return x / 2.f; }
// fonctions différentes par le nb d'arguments
int sum(int x, int y) { return x+y; }
int sum(int x, int y, int z) { return x+y+z; }
```

Conséquences :

- si la fonction n'est pas surchargée, alors le compilateur convertit implicitement le type passé en entrée dans le type du paramètre.
  Exemple : pour une fonction non surchargée `void fun(int)`, l'appel `fun(4.2f)` convertit automatiquement le paramètre en entrée en flottant.
- une surcharge entre un type et un pointeur sur ce type n'interfèrent pas l'une avec l'autre.
  Exemple : la surcharge `void fun(int*)` n'interfère pas avec `void fun(int)`.
- si la fonction est surchargée pour plusieurs types différents, alors l'appel de cette fonction avec un autre type provoque une erreur de compilation, car le compilateur ne sait pas vers quel type cet autre type doit être converti (=ambigüité).
  Exemple : pour les surcharges `void fun(int)` et `void fun(char)`, alors l'appel `fun(4.2f)` provoque une erreur d'ambigüité.

Exemple 2 :

```
int fun(int n);                    //#1
unsigned int fun(unsigned int n);  //#2
```

```
fun(4);      // appel #1
fun(4u);     // appel #2
// fun(4.f); // erreur compilation
```

Exemple 3 :

```
float sum(int x, int y);       //#1
float sum(int x, float y);     //#2
float sum(float x, float y);   //#3
```

```
sum(1, 2);       // appel #1
sum(1.f, 2.f);   // appel #3
// sum(1u, 2.f);    // erreur compilation
// sum(1, 2.0);     // erreur compilation
```

A noter qu'en C++ :

- il existe d'autres mécanismes permettant de limiter les problèmes de nommage (méthode = fonction encapsulée dans une classe, espace de nommage)
- cela ne dispense pas non plus d'adopter des conventions de nommage.

Surcharges possibles pour un paramètre de type `T` : le tableau suivant indique, pour un paramètre d'entrée (éventuellement modifié et/ou qualifié) de type `T`, quels sont les types (éventuellement modifié et/ou qualifié) qui peuvent être mis en entrée.

Types déclarés		Types surchargés			
Catég.	Type	lvalue	const lvalue	rvalue	const rvalue
value	`T`	x	x	x	x
	`const T`	x	x	x	x
lvalue	`T&`	x			
	`const T&`	x	x	x	x
rvalue	`T&&`			x	
	`const T&&`			x	x

Exemple de lecture de ce tableau :

1. un paramètre de type `T&` n'accepte en entrée que des lvalues,
2. un paramètre de type `const T&&` n'accepte en entrée que des rvalues ou des rvalues constantes.
3. un paramètre de type `const T&` accepte en entrée tout type d'argument.

Pour mieux comprendre tout ce que peut être une lvalue ou une rvalue, voir la leçon 2, section 11.2.

Les règles de surcharges sont les suivantes :

1. au sein d'une même catégorie, une surcharge est conflictuelle,
2. un paramètre de catégorie value ne peut pas être surchargé par un paramètre de catégorie `lvalue` ou `rvalue`,
3. un paramètre de catégorie `lvalue` peut être surchargé par un paramètre de catégorie `rvalue`.

Remarque : un `const T&` peut accueillir une `rvalue` sous la forme de la référence à l'espace mémoire temporaire qui stocke cette `rvalue` (et éventuellement prolonge sa durée de vie).

Utilisation des règles de surcharge

- `fun(const T)` ou `fun(const T&)` acceptent tout paramètre de type `T`.
- `fun(T)` et `fun(const T)` ambigües.

- `fun(const T&)` et `fun(T&&)` non ambigües (règle 3) : `fun(const T&)` = `lvalues` et `fun(T&&)` = `rvalues`.
- `fun(T&)` et `fun(T&&)` non ambigües (règle 3) : `fun(T&)` = `lvalues` non constantes et `fun(T&&)` = `rvalues`.

Cas des pointeurs : on ne considère ici que l'effet des qualificateurs sur la surcharge d'un pointeur. Le tableau suivant indique, pour un paramètre d'entrée (éventuellement modifié et/ou qualifié), quels sont les types qui peuvent être mis en entrée pour ce type.

argument \ entrée	`T*`	`const T*`	`T* const`	`const T* const`
`T*`	rw		rw	
`const T*`	r	r	r	r
`T* const`	rw		rw	
`const T* const`	r	r	r	r

Comment lire ce tableau ?

- si le paramètre est de type `T*`, on ne peut mettre en entrée que des pointeurs de type `T*` ou `T* const`.
- si le paramètre est de type `const T*`, on peut mettre en entrée des pointeurs avec tout variation du qualificateur `const`.

Les droits d'accès sur la mémoire pointée sont notés `rw` si le type du paramètre alloue l'accès en lecture et en écriture à la mémoire pointée, et `r` si le type du paramètre n'alloue l'accès qu'en lecture seule.

Les règles de surcharge sont les suivantes : une fonction avec un paramètre avec un accès `r` (resp. `rw`) :

1. peut être surchargée par une fonction avec un paramètre avec un accès `rw` (resp. `r`), mais pas par une fonction avec un paramètre avec un accès `r` (resp. `rw`).
2. en cas de surcharge, la fonction avec un paramètre `rw` (resp. `r`) est appelée pour les pointeurs avec un accès `rw` (resp. `r`).

Exemples :

- si la fonction `fun` possède les deux surcharges `void fun(int *)` et `void fun(const int *)`, alors :
  - ◇ `void fun(int*)` est appelé si le pointeur en entrée est de type `int*` ou `int* const`,
  - ◇ `void fun(const int*)` est appelé si le pointeur en entrée est de type `const int*` ou `const int* const`.
- il n'est pas possible de surcharger la function `void fun(int*)` avec la fonction `void fun(int* const)`.

7.2 Paramètres par défaut

En C++, il est possible de donner des valeurs par défaut aux derniers paramètres d'une fonction.

Exemple : `int fun(int a, int b=0, int c=1)` permet les appels suivants :

`x = fun(4,3,2);`	tous les paramètres sont spécifiés
`x = fun(4,3);`	appel de `fun(4,3,1)`
`x = fun(4);`	appel de `fun(4,0,1)`

Attention :

La définition de paramètres par défaut peut générer des ambiguïtés avec le nommage des fonctions.
Exemple : `int fun(int a=0)` et `int fun(void)`
Il devient alors impossible pour le compilateur de savoir quelle est la fonction qu'il faut appeler (ceci n'est donc pas autorisé).

Remarque :

la définition des paramètres par défaut est généralement déclarée dans les prototypes (voir le cours sur cette partie).

7.3 Passage des paramètres

Les paramètres d'une fonction servent à passer des valeurs à un bloc nommé (le mécanisme d'appel lui-même sera décrit plus loin).

Le passage des paramètres se déroule de la manière suivante :

- création de variables locales ayant le nom et le type des paramètres.
- initialisation de ces variables locales en utilisant les valeurs passées en paramètres à l'appel.
- exécution du code de la fonction.
- au retour de la fonction, les variables locales des paramètres sont automatiquement libérées.

Autrement dit, dans le code d'une fonction, les paramètres **sont toujours des variables locales**, et à ce titre, aucune écriture ne doit conduire à l'utiliser hors de la fonction.

Les variables locales déclarées dans la fonction ne peuvent pas occulter les paramètres de la fonction (généralement interdit par le compilateur).

a) Passage des paramètres par valeur

La passage par valeur est le mode par défaut de passage des paramètres.

A savoir,

- pour chaque paramètre, une variable locale ayant le nom du paramètre contient une copie de la valeur passée en paramètre.
- changer la valeur d'un paramètre dans le code de la fonction ne change pas la valeur passée en paramètre.

Exemple :

```
void fun(int n) {
   ...
}
// appel
fun(p);
```

```
// équivalent au code
void fun() {
   int n=p; // copie locale
   ...
}
```

Autrement dit, la modification de la valeur d'un paramètre ne modifie pas le paramètre passé à la fonction.

b) Passage des paramètres par référence à une lvalue

Ce mode n'est possible qu'en C++.

Rappel : une référence (à une lvalue) est une façon de donner un autre nom à une variable (= une seule variable avec deux noms différents), cf leçon 2.

Conséquences :

- Lorsqu'un paramètre est déclaré comme étant une référence, le paramètre devient un alias de la variable passée en référence.
- Donc, la variable passée en paramètre et le paramètre représentent le même objet en mémoire.
- Modifier le paramètre dans la fonction, revient donc à modifier la variable passée en paramètre.

Exemple :

```
void fun(int &n) {
   ...
}
```

```
// fun: n autre nom pour p
fun(p);
// fun(4); // erreur compil.
```

Inconvénient :

Il n'est pas possible de passer des valeurs constantes en paramètre à une fonction sauf si l'on utilise une référence constante (`const int&` dans l'exemple ci-dessus, voir aussi plus loin), mais dans ce cas, impossible de modifier la valeur.

c) Passage des paramètres par référence à une rvalue

Ce mode n'est possible qu'en C++11.

Rappel : une référence à une rvalue est une façon de faire référence à un résultat temporaire, voir la leçon à ce sujet.

Conséquences : permet d'écrire une fonction qui va être exécutée spécifiquement sur les rvalues.

Exemple :

```
void fun(int &n)
// #1 (ref. lvalue)
void fun(int &&n) // #2 (ref. rvalu
int p=4;
```

```
fun(p);   // appel #1
fun(4+p); // appel #2
fun(4);   // appel #2
```

Remarque :

Ne pas utiliser ce mode de transmission de paramètres dans un premier temps, tant que ce sujet n'a pas été abordé en profondeur, afin d'en comprendre l'intérêt.

d) Passage des paramètres par pointeur

Rappel : un pointeur est une adresse mémoire, généralement utilisée pour indiquer où est stockée une objet dans la mémoire.

Pour passer un paramètre par pointeur :

- définir le type du paramètre comme étant un pointeur.
- à l'appel de fonction, passer l'adresse du paramètre (opérateur &) ou directement le pointeur s'il pointe sur une zone de mémoire allouée.

- dans la fonction, utiliser le pointeur pour modifier directement le paramètre en écrivant directement à l'adresse stockée dans le pointeur (opérateur *).

Exemple :

```
void fun(int *p) {
   ...
   *p = val;
}
```

```
int  a=4;
fun(&a); // ici a=val
int  *b = new int(4);
fun(b);  // ici *b=val
```

Exemple : analyse

```
// Exemple fictif
int  a=4;
// a stocké en
// &a = 0x1234
fun(&a);
// appel fun(0x1234)
```

```
void fun(int *p) {
  // ici: p=0x1234
  ...
  // écriture en 0x1234
   *p = val;
}
```

Autrement dit, le passage du paramètre reste par valeur, mais on utilise le pointeur pour aller modifier directement la variable dont on passe l'adresse en paramètre.

Inconvénient : il est nécessaire de tester si le pointeur passé n'est pas nul afin d'être sûr d'écrire à une adresse valide.

Note : ce point sera revu lors de l'étude des pointeurs et de l'allocation dynamique.

e) Constant

Le modificateur `const` peut également être utilisé sur les paramètres d'une fonction.

Un `const` sur un paramètre signifie que ce paramètre est une constante dans le code de la fonction.

- passage par valeur : le paramètre local est une constante.
- passage par référence : la référence ne peut pas être modifiée, permet également de passer à la fonction des constantes ou des temporaires.
- passage par pointeur : l'objet pointé ne peut pas être modifié.

Exemple : analyse

```
int funV(const int x);
int funR(const int &x);
const int a=4;
int b=5;
```

```
funV(6);    funR(6);    // OK!
funV(a+6);  funR(a+6);  // OK!
funV(a);    funR(a);    // ok
funv(b);    funR(b);    // ok
```

Quel intérêt d'ajouter `const` à un paramètre ?

- par valeur : il s'agit de forcer une règle d'usage pour un paramètre : il est constant, et changer sa valeur dans le code de la fonction est une erreur.
- par pointeur ou référence : éviter la copie et garantir à l'utilisateur que le paramètre passé ne sera pas modifié par l'appel de la fonction.
- inutile pour une référence à une rvalue (expliqué plus tard).

7.4 Mécanisme d'appel des fonctions

Environnement local d'une fonction = mémoire nécessaire pour les paramètres et les variables locales.

L'environnement local d'une fonction permet son exécution de la fonction de **façon indépendante** de la fonction qui l'a appelée.

Le mécanisme d'appel d'une fonction est le suivant :

- allocation de l'environnement local (= réservation de la mémoire nécessaire).
- initialisation des variables associés aux paramètres en utilisant les valeurs passées en paramètres lors de l'appel.
- exécution de la fonction dans l'environnement local.
- au retour de la fonction, copie de la valeur de retour au lieu d'appel de la fonction.
- libération de l'environnement local (paramètres et variables locales).

Toute fonction utilise pour cette méthode pour s'exécuter.

7.5 Pile d'appel des fonctions

Pile = tas symbolique utilisé pour empiler des données sur le principe du premier entré, dernier sorti.

L'exécution d'un code s'effectue donc par l'exécution de la fonction `main` de la façon suivante :

1. exécution séquentielle du code de la fonction.
2. si on rencontre l'appel d'une fonction alors
   a) on empile l'environnement courant.
   b) création et initialisation de l'environnement local de la fonction.
   c) aller en 1.
3. si `return v`
   a) libération de l'environnement local.
   b) on dépile l'environnement précédent.
   c) on continue le code en utilisant la valeur `v` calculée.

Un environnement d'exécution contient le point courant d'exécution (i.e. la ligne, l'environnement local alloué pour cet appel de la fonction).

En un point quelconque de l'exécution d'un code, on dispose de la pile complète des environnements d'exécution contenant :

- le point exact d'exécution du code,
- la suite des fonctions ayant conduit à ce point d'exécution,
- la valeur de tous les paramètres d'appel des fonctions associées,
- la valeur de toutes les variables locales au point d'exécution courant pour chacun.

A noter que si une même fonction est appelée plusieurs fois dans cette pile, il y a autant d'environnement d'exécution que d'instances de l'appel (exemple : les fonctions récursives).

Cet pile d'exécution peut être étudiée en cours d'exécution du programme grâce à l'utilisation d'un débogueur.

7.6 Inlining

La conséquence de ce fonctionnement est qu'un appel de fonction a un coût : les suites de (empilement / création / destruction / dépilement) prennent du temps.

Pour les fonctions simples, ce coût n'est pas négligeable devant le temps d'exécution de la fonction.

Est-il alors intéressant d'écrire des usines à gaz ? (= fonction longue exécutant de nombreuses tâches différentes).

Évidemment, non :

- lors d'une compilation en mode optimisée, les appels de fonction disparaissent et le code des fonctions les plus simples est directement exécuté au lieu d'effectuer un appel.
- il est aussi possible d'aider le compilateur en lui signalant les fonctions pour lesquelles il est préférable de le faire : ce procédé s'appelle l'inlining.

a) Macros du précompilateur

En C, le passage des fonctions en ligne est effectuée en utilisant les macros du précompilateur.

- Une macro est une fonctionnelle du précompilateur permettant de générer du code.
- **Exemples :**
  - ◇ `#define max(a,b) ( (a)>(b) ? (a) : (b) )`
    effectue un remplacement **impératif** de la fonction `max` par le code indiqué (noter la présence des parenthèses).
  - ◇ une macro peut être inefficace `max(sin(x),cos(x))`,
  - ◇ une macro peut être dangereuse :
    `#define mul(a,b) a*b` avec `mul(3+2,4)`
    `#define vec3(x) {x,x,x}` avec `vec3(i++)`
- Les arguments d'une macro n'est pas typée (pas de possibilité de vérifier le type des paramètres).

En raison de ces nombreux problème, le C++ introduit de nouveaux outils.

b) Nouveaux outils du C++

- les fonctions `inline` : pour écrire de petites fonctions courte destinée à être remplacée par leurs codes.
- les expressions constantes `constexpr` (C^{++}_{11}) : pour écrire des fonctions dont le résultat peut être évalué à la compilation lorsqu'on leurs passe des arguments constants.
- les fonctions génériques (`template`) : pour écrire des fonctions dont le type est générique (également pour les définitions de types).
  Abordé dans une leçon ultérieure.
- les macros du préprocesseur dans le cas où aucun des outils ci-dessus ne permettrait d'obtenir le résultat recherché.
- les lambdas expressions (définition au vol d'une fonction, voir les chapitres sur les fonctionnelles).

c) Fonctions inline

On utilise le mot-clé `inline` pour signifier au compilateur que cette fonction doit **si possible** être remplacée par son code.

Exemple :

```
inline int max(int a,int b) { return (a>b?a:b); }
inline float mul(float x, float y) { return x*y; }
float v=mul(3+2,4);
int   u=max(4,int(v));
```

Notes :

- suggestion et non obligation pour le compilateur (il y a des alternatives),
- les types sont vérifiés à l'appel,
- pas de risque à l'évaluation,
- `max` peut sembler maintenant trop typé (ne fonctionne plus avec tous les types pour lesquels l'opérateur > est défini). Utiliser les `template`s dans ce cas.
- elle ne peut pas être virtuelle.

d) Fonction `constexpr`

- `constexpr` sur une fonction implique `inline` (donc, non virtuelle),
- son type de retour doit être littéral, ses arguments doivent être des littéraux.
- la totalité de la fonction doit pouvoir s'exécuter dans le `return` (*i.e.* est le résultat de l'évaluation d'une seule expression). Cette limitation sera levée dans C_{15}^{++}.
- au moins une invocation de la fonction doit pouvoir conduire à expression constante (i.e. retourne une expression constante et/ou n'appelle que des fonctions `constexpr`).

Exemple :

```
// fonction factorielle constexpr
constexpr int fact(int n) {
   return n <= 1? 1 : (n * fact(n - 1)); }
```

```
// évalué à la compilation: résultat = 24 true
constexpr int k1 = 4;
constexpr int v1 =  fact(k1);
// évalué à l'exécution
volatile int k2 = 5;
int v2 = fact(k2);
```

8 Structuration et compilation d'un code

8.1 Principes

Principes conducteurs de structuration d'un code C++ :

- similaire à un code C.
- tout ce qui est utilisé doit avoir été défini **avant** utilisation.
  sinon, échec de la compilation.

- l'ordre d'utilisation ne devrait pas dépendre de l'ordre de définition.

Problèmes :

- semble obliger à un ordre de définition précis des fonctions (plus une fonction appelle d'autres fonctions, plus elle semble devoir être définie tardivement).
- que faire si une fonction `fun1()` appelle une fonction `fun2()`, et que cette même fonction `fun2()` appelle aussi la fonction `fun1()` (cas d'une référence circulaire).
- dans quel ordre effectuer les différentes définitions ?

8.2 Prototypage

Le prototypage d'une fonction consiste à définir une fonction sans donner son code.

Prototype d'une fonction = façon de dire :

- la fonction a tels paramètres et renvoie tel type de données,
- avec l'obligation de définir le code de la fonction plus loin
  si ce n'est pas fait ET que la fonction est utilisée, l'édition de lien échoue.

Exemple :

la fonction `int fun(int a,float b) { /* code */ }`

a pour prototype `int fun(int,float);` (sans le code).

Le nom des variable peut aussi être laissé.

Remarque : dans le cas d'une fonction avec des valeurs de paramètres par défaut, les valeurs doivent être données dans le prototype. **exemple :** `int fun(int,float=3.14f);`

Note : dans un bibliothèque, si seul le prototype est défini, mais le code n'est pas donné, le code compile mais échoue à l'édition des liens.

8.3 Ordre de définition

L'ordre de définition qu'il est préférable d'utiliser dans un code simple est le suivant :

- **constantes**
- **types**
- **prototypes**.
- **fonctions** avec leurs codes.

Une définition dans cet ordre permet de faire en sorte que :

- pour les constantes, on les trouve toutes facilement en début de code. Elles ne dépendent de rien, si ce n'est d'autres constantes.
- pour les types, ils ne dépendent que des constantes ou d'autres types.
- pour les prototypes, ils ne dépendent que des types et des constantes.
- pour les fonctions, elles peuvent, en ce point du code, être définies dans n'importe quel ordre.

8.4 Modules

a) Idée

Dès que le code n'est pas élémentaire, il est découpé est unités fonctionnelles appelées **module**s.

L'intérêt de la modularisation est que les modules peuvent être réutilisables dans différents programmes ; ils doivent d'ailleurs être écrit dans cet objectif.

Exemple : pour un code qui utilise de l'algèbre linéaire, on va utiliser les modules : vector, matrix, ALresolve, ... où :

- vector : module de définition du type vector et de manipulation de base des vecteurs (construction, initialisation, opérations vectorielles, produit scalaire, ...)
- matrix : module de définition du type matrix et de manipulation élémentaire (construction, initialisation, opération matricielle élémentaire).
- ALresolve : module de résolution de système linéaire (implémentation de différentes méthodes)

Problème : comment définir ces morceaux et les assembler ?

b) Définition

Un module est constitué de la façon suivante :

- un fichier d'en-tête (extension `.h`) qui contient la déclaration de l'ensemble des constantes, types et fonctions définies dans le module
- un ficher source (extension `.cpp`) qui contient la définition de l'ensemble des fonctions du module (*i.e.* leurs prototypes).

L'assemblage des modules s'effectue de la façon suivante :

- dans tout code qui utilise un module, inclure le fichier d'en-tête associé au module.
  voir ci-après pour le code type d'un fichier d'en-tête d'un module et comment l'inclure.
- lors de la compilation, tous les codes des modules utilisés doivent être compilés, puis assemblé.
  ceci s'effectue facilement en écrivant un `makefile` et en utilisant la commande `make`, voir ci-après.

c) Inclusion de modules externes

L'inclusion du fichier d'en-tête `toto.h` dans un code `tutu.c` s'effectue de la manière suivante :

- `#include "toto.h"` : pour inclure les déclarations d'un module personnel.
  ici `toto.h` est placé dans le même répertoire que `tutu.c`.
  Des options de compilation peuvent changer ce comportement.
- `#include <toto>` : pour inclure les déclarations d'un module système.
  Les fichiers d'en-tête des modules système sont placés dans `/usr/include` sur un système unix.

`#include` signifie littéralement coller le texte du fichier qui suit à cet endroit du code.

Problème :

Les différentes inclusions des fichiers d'en-tête peuvent provoquer l'inclusion multiple d'un même fichier d'en-tête.
L'inclusion multiple d'un fichier d'en-tête provoque la redéfinition de son contenu (=erreur de compilation).

La solution consiste à utiliser des commandes du précompilateur de manière à empêcher les inclusions multiples.

Le code d'un fichier d'en-tête (ici `toto.h`) doit être le suivant :

```
#ifndef _TOTO_H
#define _TOTO_H
// mettre ici les déclarations
#endif
```

Signification :

- Si le symbole `_TOTO_H` n'est pas défini, alors on le définit, et on inclut les déclarations du module `toto.c` (à savoir le contenu du `toto.h`).
- Si le symbole `_TOTO_H` est défini, alors on fait rien (ceci signifie qu'il a déjà été inclus).

L'ensemble de la bibliothèque standard du `C` (stdio, stdlib, ...) est utilisable en `C++` de la manière suivante :
pour inclure `stdio.h` faire `#include <cstdio>`

Comme indiqué en introduction, de nombreuses raisons font qu'un code `C` peut ne pas être compilé avec le compilateur `C++`.

Mais la compilation ne pose en général pas de problème sous les réserves suivantes :

- éviter de mélanger des `.c` et `.cpp` dans un même projet.
  à savoir linker des `.c` compilés avec un compilateur `C` et des `.cpp` avec un compilateur `C++` (source de problème).
- **le plus simple :** renommer les extensions `.c` en `.cpp` et tout compiler avec le compilateur `C++`.

Inversement, pour écrire en `C++` appelable depuis le `C`, les fonctions appelables de la bibliothèque doivent être définie dans des blocs `extern "C" { ... };`

8.5 Retour sur les classes de stockage

On considère l'exemple suivant :

Exemple :

```
// unitA.cpp
int  x=4;
int fun2() { int y; }
// depuis unitB.h
extern int fun();
```

```
// unitB.cpp
int fun() { static int b=0; }
// depuis unitA.h
extern int fun2();
extern int a;
```

Analyse : classe de stockage par unité de traduction

- `x` : variable globale
  définie dans `unitA.cpp` = statique, lien interne.

déclarée `unitB.cpp` = statique, lien externe.
- `y` : variable locale de `fun2` dans `unitA.cpp`
  dans `unitA.cpp` = automatique, pas de lien.
  dans `unitB.cpp`, non accessible.
- `b` : idem `y` mais en inversant le rôle des unités de traduction.
- `fun2` : fonction
  définie dans `unitA.cpp` = statique, lien interne.
  déclarée `unitB.cpp` = statique, lien externe.
- `fun` : idem `fun2` mais en inversant le rôle des unités de traduction.

Attention à l'effet des modificateurs `const`, `static` et `extern` sur le lien des objets déclarés en global.

- `static` devant la déclaration d'une variable globale ou d'une fonction implique que l'objet a un lien interne.
  - ◇ pour une variable globale, cela signifie que si la variable doit être initialisée, elle doit l'être dans cette unité de traduction,
  - ◇ pour une fonction, cela signifie que si la fonction doit être définie, elle doit l'être dans cette unité de traduction,

  L'objet ne peut donc pas être spécifié `extern` en même temps.
- `extern` devant la déclaration d'une variable globale ou d'une fonction implique que l'objet a un lien externe, donc déclaré dans l'unité de traduction courante et définie dans une autre unité de traduction.
  La présence d'une définition dans l'unité de traduction courante provoque donc une erreur.
- par défaut, une fonction définie sans modificateur est `extern` sauf si les définitions suivantes impliquent le contraire (à savoir, la fonction est définie).
- par défaut, les objets déclarés comme `const` et qui ne sont pas explicitement déclarés `extern` ont un lien interne.

8.6 Compilation

a) Compilation élémentaire

La manière la plus aisée de construire un exécutable à partir d'un code source est d'utiliser la commande `make`.

Elle contient les règles de construction d'un exécutable à partir de l'extension du fichier (taper `make -p` pour voir les règles prédéfinies).

Les extensions à utiliser sont les suivantes :

- `.h` pour un fichier d'entête,
- `.cpp` pour un code écrit en langage C++.

Lorsque la totalité d'un code est dans un seul source (par exemple `toto.cpp`), la compilation s'obtient avec :

```
make toto
```

et produit un exécutable nommé `toto`.

b) Compilation modulaire

On a maintenant un code constitué d'un code principal `main.cpp` et de deux modules (`fun1.cpp` et `fun2.cpp`).

La construction automatique de l'exécutable est simplifiée et automatisée en créant un fichier `makefile` décrivant ses différentes parties, et contenant au minimum :

```
OBJS=main.o fun1.o fun2.o
prog: $(OBJS)
<tab>g++ $(OBJS) -o main
```

où <tab> est une tabulation.

La compilation s'effectue en tapant `make` ou `make prog`.

Attention : (redite) ne pas mélanger fichier C et C++ dans un même projet, sauf à renommer les extensions C en C++.
Il est également possible de changer les options de compilation dans un `makefile` en redéfinissant les symboles suivants :

- `CPPFLAGS` : options de compilation (`-g -Wall` en mode débogue, `-O3` en mode optimisé). ajouter `-std=c++11` pour compiler du code C_{11}^{++}.
- `LDLIBS` : bibliothèques avec lesquelles compiler (`-lm` pour la bibliothèque mathématique, ...).
- `CXX` : nom du compilateur à utiliser (en général, `g++` pour le compilateur C++).

Il est également possible d'ajouter d'autres règles de construction ou des règles effectuant des tâches courantes.

Exemple : à exécuter avec `make clean`

```
clean:
<tab>rm -f *.o
```

c) Dépendances

Problème avec la compilation séparée : si le module A utilise le module B, la modification des structures ou des prototypes de B doit nécessairement entraîner la recompilation de A.

On appelle ce phénomène **dépendance**.

Des règles de dépendance peuvent être ajoutés dans le `makefile` :

- soit manuellement, en ajoutant au `makefile` des lignes du type :
  `fun2.o: fun1.h`
  signifiant, si le contenu de `fun1.h` change, alors `fun2.o` doit être reconstruit. Cette règle devrait être entrée pour tout code incluant l'en-tête d'un module dont il dépend.
- soit automatiquement en utilisant la commande `makedepend` (si elle est installée).
  `makedepend fun2.cpp` (dans une console) permet d'ajouter automatiquement les règles de dépendance de `fun2.cpp` dans le `makefile`.

Note : `makedepend` est généralement dans le package `xutils-dev` ou `imake`. A installer au besoin.

d) `makefile` avancé

On propose le `makefile` type suivant :

```
# partie du makefile à modifier
EXE=prog
SRCS=main.cpp fun1.cpp fun2.cpp
CPPFLAGS=-g -Wall -Wconversion -std=c++11
LDLIBS=
CXX=g++
# partie à ne pas toucher
OBJS=$(SRCS:.cpp=.o)
$(EXE): depend $(OBJS)
<tab>$(CXX) $(OBJS) $(LDLIBS) -o $@
depend:
<tab>@makedepend -Y $(SRCS) 2> /dev/null
clean:
<tab>rm -f  $(OBJS)
all: clean depend $(EXE)
```

qui permet de disposer des commandes suivantes pour un code :

- `make` pour compiler le code.
- `make depend` pour rechercher et générer les dépendances.
- `make clean` pour effacer les fichiers objets.
- `make all` pour effacer tous les fichiers objets, reconstruire les dépendances et recompiler le projet.

Note : `c++0x` sur version de `g++` plus ancienne.

e) Intérêt de l'utilisation de `make`

On automatise la tâche de la compilation :

- en ne recompilant que les codes qui ont changés : le `makefile` détecte automatiquement les codes modifiés. La compilation de projet avec un nombre important de modules est plus rapide.
- en recompilant les codes qui dépendent de modules dont la définition a changé : grâce à l'utilisation et la génération de dépendance.
- en assemblant automatiquement l'ensemble des modules et en construisant de l'exécutable.
- en effaçant les fichiers objet (ou nettoyer un projet) sans risquer d'effacer des fichiers autres que ceux souhaités.

`make` est donc un outil important permettant de construire facilement et correctement des projets complexes.

8.7 Outils de développement

Un compilateur peut être un outil précieux pour trouver des erreurs autres que des erreurs syntaxiques :

- le compilateur C/C++ avec l'option `-Wall` affiche les avertissements associés à des comportements à risque (problème de portabilité, ...).
- l'utilisation d'un outil de vérification statique (outil effectuant une analyse plus poussé du code qu'un compilateur et signalant des erreurs d'ordre sémantique) est vivement conseillé.
  Exemple : `lint`, `splint`, `cppcheck`

Dans tous les cas,

- **aucun warning ne doit rester incompris**.
- **aucun warning ne doit rester** : le code doit être corrigé afin que tous les warnings disparaissent.

L'IDE (integrated development environment) très vivement conseillé pour ce cours est `CLion` de JetBrain.

Liens vers les pages permettant d'obtenir :

- une licence étudiant : `https://www.jetbrains.com/student/`
- la dernière version : `https://www.jetbrains.com/clion/`

Cet IDE a notamment les fonctionnalités suivantes :

- éditeur intelligent : complétion automatique sur les noms, les champs, les méthodes ; formatage automatique, ...
- analyse du code au vol signalant les erreurs, et proposant des corrections automatiques
- navigation facile dans le code entre les symboles, les classes, leurs déclarations, leurs codes, ...
- génération de code et refactoring,
- exécution et debugging inclus dans l'éditeur (intégration de `gdb` et `lldb`),
- analyse dynamique de la mémoire à l'exécution (intégration de `Valgrind` et `Google Sanitizers`),
- utilisation sous-jacente de `cmake` pour les projets,
- intégration d'outils de développement logiciel : test unitaire (Catch, Boost, ...), documentation (Doxygen), versionnage de code (CVS, Git, Subversion, ...).

Conclusion

Nous avons vu dans ce chapitre l'ensemble des outils élémentaires :

- quels sont les types qui peuvent être utilisés pour définir des objets (élémentaires, énumérés, agrégés) et qu'il est possible de définir de nouveaux types.
- que ces types peuvent être modifiés ou qualifiés afin de changer le comportement d'une variable,
- qu'une attention particulière doit être apportée aux variables numériques afin d'éviter des conversion inutiles ou inadéquates,
- quels sont les différents types de variables et de constantes qui peuvent être utilisées dans l'écriture d'un code, ainsi que leurs propriétés (classe de stockage, localité, portée, ...)
- quels sont les ajouts du C++ dans la définition des fonctions, le passage des paramètres, et la possibilité de définir des fonctions en ligne,
- comment fonctionne le mécanisme d'appel des fonctions afin de le comprendre et prendre conscience de son coût,
- comment structurer un code C++, le compiler, décomposer un programme en module, et écrire des `makefile`s afin de compiler le code.

Les chapitres suivants (chapitre 2 sur l'encapsulation, et 3 sur héritage, virtualité et polymorphisme) précise les possibilités des types agrégés.

Le chapitre 8 complètera la partie sur les fonctions en indiquant tous les outils fonctionnels dont l'on

dispose en C++.

Attention, pour apprendre un langage de programmation, il est **nécessaire de programmer**.

- **intégrer la syntaxe** ne peut se faire sans pratique.
  aux évaluations, une réponse dont la syntaxe est manifestement fausse sera comptée comme fausse.
- **intégrer le cours** ne peut se faire sans avoir programmé par vous-même.
- maitriser le C++ est difficile et ne peut se faire sans vous investir dans ce cours.